

# 目 录

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

## 第一部分 C语言基础

第一章 词 法 .....	1
1.1 标识符 .....	1
1.2 关键字 .....	2
1.3 常 量 .....	2
1.3.1 整数常量 .....	2
1.3.2 浮点数常量 .....	3
1.3.3 枚举常量 .....	4
1.3.4 字符常量 .....	4
1.4 字符串 .....	5
1.5 运算符 .....	5
1.5.1 单目运算符 .....	7
1.5.2 双目运算符 .....	8
1.5.3 三目运算符 .....	9
1.5.4 赋值运算符 .....	10
1.5.5 逗号运算符 .....	10
1.5.6 函数参数运算符 .....	11
1.5.7 数组下标运算符 .....	11
1.5.8 结构/联合成员运算符 .....	11
1.5.9 结构/联合指针运算符 .....	11
1.6 分隔符 .....	11
1.6.1 方括号分隔符 .....	12
1.6.2 圆括号分隔符 .....	12
1.6.3 花括号分隔符 .....	12
1.6.4 逗号分隔符 .....	13
1.6.5 分号分隔符 .....	13
1.6.6 冒号分隔符 .....	13
1.6.7 省略号分隔符 .....	13
1.6.8 星号分隔符 .....	14
1.6.9 等号分隔符 .....	14
1.6.10 预处理器伪指令符 .....	14
1.7 空白符(nul) .....	14

第二章 说 明 .....	15
2.1 对象与左值 .....	15
2.1.1 对 象 .....	15
2.1.2 左 值 .....	16
2.2 变量说明 .....	16
2.3 数组说明 .....	18
2.3.1 一维数组, 一维指针数组, 一维数组指针 .....	19
2.3.2 多维数组, 多维指针数组, 多维数组指针, 多重指针 .....	20
2.4 结构说明 .....	20
2.4.1 原型法 .....	21
2.4.2 类型别名法 .....	21
2.4.3 关于结构的讨论 .....	22
2.4.4 位段结构 .....	23
2.5 联合说明 .....	23
2.6 函数说明 .....	24
第三章 语 句 .....	27
3.1 语 句 .....	27
3.2 表达式 .....	27
3.3 表达式语句 .....	27
3.4 复合语句 .....	28
3.5 循环语句 .....	29
3.5.1 for 循环语句 .....	29
3.5.2 while 循环语句 .....	30
3.5.3 do while 循环语句 .....	31
3.6 条件语句 .....	32
3.6.1 一般条件语句 .....	32
3.6.2 嵌套条件语句 .....	33
3.6.3 多选一条件语句 .....	34
3.7 开关语句 .....	36
3.8 间断语句 .....	38
3.9 接续语句 .....	39
3.10 跳转语句 .....	39
3.11 返回语句 .....	40
3.12 空语句 .....	42
第四章 函数及函数库 .....	44
4.1 前 言 .....	44

4.2 单文件程序(一)——字符串处理	44
4.3 单文件程序(二)——二维数组	46
4.4 多文件程序——台式计算器逆波兰算法的实现	47
4.5 关于函数参数值的传送问题	51
4.6 主函数	52
4.7 C语言的函数库	54
4.8 头文件	54
4.9 分类库函数	56
4.9.1 归类函数	56
4.9.2 转换函数	56
4.9.3 目录控制函数	57
4.9.4 诊断函数	57
4.9.5 图形函数	57
4.9.6 内部函数	58
4.9.7 输入输出函数	58
4.9.8 各类接口函数(dos, bios, 8086)	60
4.9.9 串与内存块操作函数	61
4.9.10 数学函数	62
4.9.11 动态内存管理函数	63
4.9.12 杂项函数	63
4.9.13 进程控制函数	63
4.9.14 窗口文本显示函数	64
4.9.15 日期时间函数	64
4.9.16 变参数表函数	64
4.10 全局变量	65
4.10.1 argc	65
4.10.2 argv	65
4.10.3 ctype	65
4.10.4 daylight	65
4.10.5 directvideo	65
4.10.6 environ	66
4.10.7 error, _doserrno, sys_errlist, sys_nerr	66
4.10.8 fmode	67
4.10.9 heaplen	67
4.10.10 _new_handler	68
4.10.11 _osmajor, _osminor	68
4.10.12 _ovrbuffer	68
4.10.13 _psp	68
4.10.14 _stklen	68

4.10.15	timezone .....	69
4.10.16	tzname .....	69
4.10.17	_version .....	69
4.10.18	_wscroll .....	69
4.10.19	_8087 .....	69
<b>第五章</b>	<b>预处理器 .....</b>	<b>71</b>
5.1	前 言 .....	71
5.2	包含文件伪指令 .....	71
5.3	伪指令宏 .....	72
5.3.1	简单宏 .....	72
5.3.2	参数宏 .....	72
5.3.3	宏释放 .....	73
5.3.4	条件宏定义 .....	73
5.3.5	预定义宏 .....	74
5.3.6	宏体中使用转义符#和合并符## .....	74
5.4	条件编译伪指令 .....	75
5.5	#pragma 伪指令 .....	75
5.6	#line 伪指令 .....	75
5.7	#error 伪指令 .....	76
<b>第二部分 C51(8051 用 8 位嵌入式 C 语言)</b>		
<b>第六章</b>	<b>C51 前言 .....</b>	<b>78</b>
<b>第七章</b>	<b>C51 说明 .....</b>	<b>79</b>
7.1	C51 简单变量说明 .....	79
7.1.1	类型说明符 bit .....	80
7.1.2	预定义特殊功能寄存器说明符 sfr 和 sfr16 .....	80
7.1.3	预定义特殊功能寄存器位说明符 sbit .....	81
7.1.4	在 bdata RAM 空间定义位变量(借用位类型符 sbit) .....	81
7.2	C51 复合变量说明 .....	82
7.3	C51 指针变量说明 .....	82
7.3.1	通用指针 .....	83
7.3.2	抽象指针——匿名指针 .....	84
7.3.3	指针可用运算符 .....	85
<b>第八章</b>	<b>C51 存储模式 .....</b>	<b>86</b>
8.1	C51 三种存储模式 .....	86

8.2 C51 内部对数据和函数的组织规范	87
8.2.1 标识符改大写字符和函数换名	87
8.2.2 全局变量存放的段名规定	87
8.2.3 函数的段名	87
8.2.4 函数的参数传送规则	88
8.2.5 重入栈的有关规定	89
8.2.6 函数返回值的规定	89
<b>第九章 C51 函数及库函数</b>	<b>90</b>
9.1 函数说明	90
9.2 函数被修饰使用指定的寄存器组	91
9.3 函数被修饰为中断函数	92
9.4 函数被修饰为重入函数	93
9.5 函数被修饰为使用指定的存储模式	94
9.6 C51 与 PL/M51 函数的交叉调用	95
9.7 C 与汇编函数的交叉调用	95
9.8 内部函数	100
9.8.1 左移多位函数	100
9.8.2 右移多位函数	101
9.8.3 空操作函数	101
9.8.4 位测试函数	102
9.9 抽象数组(绝对地址存取)——absacc 库函数	102
9.10 C51 库函数介绍	103
<b>第十章 C51 SFR 头文件和配置文件</b>	<b>107</b>
10.1 特殊功能寄存器头文件	107
10.2 C51 配置文件	107
10.2.1 STARTUP.A51 文件	108
10.2.2 INIT.A51 文件	108
10.2.3 PUTCHAR.C 文件	109
10.2.4 GETKEY.C 文件	109
<b>第十一章 C51 预处理器伪指令</b>	<b>110</b>
<b>第十二章 C51 编译命令行控制选项和控制伪指令</b>	<b>111</b>
12.1 简介	111
12.2 编译命令行	111
12.2.1 一次性使用编译控制伪指令	112
12.2.2 可多次使用编译控制伪指令	118

<b>第十三章 C51 及 L51 使用方法</b>	122
13.1 C51 的使用环境	122
13.2 C51 安装	122
13.3 编译方法	123
13.4 C51 支持的文件名和设备名	123
13.5 错误号	123
13.6 连接/定位方法	124
13.7 连接控制选项	125
13.7.1 一般的连接控制选项	125
13.7.2 特殊的连接控制选项	126
13.8 定位控制选项	127
13.9 映像列表文件控制选项	129
13.10 连接/定位命令	129
13.11 特殊连接控制选项示例	131
13.12 使用 C51 和 L51 的完整示例	134
13.12.1 多模块编程	135
13.12.2 多模块编译	135
13.12.3 多模块连接定位	137

### 第三部分 XAC(80C51XA 用 16 位嵌入式 C 语言)

<b>第十四章 XAC 说明</b>	143
14.1 XAC 变量说明	143
14.1.1 XAC 一般变量说明	143
14.1.2 绝对变量与 SFR	147
14.1.3 位变量与可位寻址 SFR	147
14.2 XAC 数组说明	148
14.3 XAC 结构说明	148
14.4 XAC 联合说明	148
14.5 XAC 函数说明	148
14.5.1 XAC 一般函数说明	149
14.5.2 XAC banked 中断函数说明	149
14.5.3 中断向量表(ROM 向量表)的添写	150
14.5.4 中断接管与 RAM 向量表	152
<b>第十五章 XAC 编译器内部管理规范和约定</b>	154
15.1 XAC 标准程序子段(psect)	154
15.2 XAC 有关寄存器的约定	155



15.3	XAC 有关参数传送和函数返回的约定 .....	155
15.4	XAC 关于函数的签字 .....	156
15.5	XAC 有关存储器的约定 .....	156
15.6	XAC 的存储模式 .....	156
15.7	XAC 关于运行时启动模块的规定 .....	158
15.8	XAC 上电子程序 .....	158
15.9	XAC 标准启动模块的编程 .....	158
15.9.1	连接器定义符号名 .....	158
15.9.2	bss 和 rbss 清零程序 .....	159
15.9.3	data 和 rdata 复制程序 .....	159
15.10	XAC 定制的启动模块 .....	159
15.10.1	手工优化代码 .....	159
15.10.2	定制启动模块的编写 .....	160
15.10.3	关于版权信息 .....	160
第十六章	XAC 的混合编程和函数库 .....	161
16.1	C 语言与汇编语言混合编程 .....	161
16.1.1	C 与汇编函数的交叉调用 .....	161
16.1.2	在线嵌入汇编指令段 .....	162
16.2	XAC 运行时间库函数 .....	162
16.2.1	标准输入输出库函数及用户的定制 .....	162
16.2.2	XAC 库函数汇总 .....	163
16.2.3	XAC 库管理器实用程序 .....	166
第十七章	XAC 编译器 .....	168
17.1	编译命令行控制选项 .....	168
17.1.1	-A(指定 ROM 和 RAM 定位地址) .....	169
17.1.2	-AAHEX(指定按美国自动化符号格式生成 HEX 文件) .....	169
17.1.3	-AV((指定符号文件用 Avocet 风格) .....	170
17.1.4	-BIN(指定生成二进制输出文件) .....	170
17.1.5	-BI(指定选用大存储模式) .....	170
17.1.6	-Bm(指定选用中存储模式) .....	170
17.1.7	-Bs(指定选用小存储模式) .....	170
17.1.8	-C(只翻译到目标文件) .....	171
17.1.9	-CR(生成交叉访问表) .....	171
17.1.10	-CLIST(生成 C 列表文件) .....	171
17.1.11	-D(定义宏) .....	171
17.1.12	-DOUBLE(起用 IEEE64 位 DOUBLE 变量) .....	172
17.1.13	-E(编译器使用 editor 格式的错误信息) .....	172

17.1.14	-E(编译器错误信息重定向到指定文件)	172
17.1.15	-H(生成汇编级符号文件)	172
17.1.16	-I(指定附加的搜索头文件的路径)	173
17.1.17	-L(指定附加的扫描库)	173
17.1.18	-L- (指定传递给 LINKER 的控制选项)	173
17.1.19	-M(生成映像文件)	173
17.1.20	-MOTOROLA(生成 Motorola S-Record 格式的 HEX 文件)	174
17.1.21	-N(指定标识符有效字符长度)	174
17.1.22	-O(启动优化)	174
17.1.23	-O(指定输出文件)	174
17.1.24	-OMF51(指定生成 OMF51 格式的输出文件)	174
17.1.25	-PROTO(指定生成包括 ANSI 和 K&R 风格的函数原型文件)	174
17.1.26	-PSECTMAP(程序段映像表)	174
17.1.27	-S(编译生成汇编源文件)	175
17.1.28	-STRICT(严格遵守 ANSI 标准)	175
17.1.29	-TEK(编译生成 Tektronics HEX 文件)	175
17.1.30	-U(解除宏定义)	176
17.1.31	-UBROF(指定生成 UBROF 格式的输出文件)	176
17.1.32	-UNSIGNED(指定 unsigned char 为 char 的缺省类型)	176
17.1.33	-V(详示编译命令)	176
17.1.34	-W(设置告警级别)	176
17.1.35	-X(去除局部符号)	176
17.1.36	-Zg(启动全局优化)	176
17.2	编译器输出文件格式	176
17.3	编译器生成的符号文件	177
17.4	CREF 生成交叉访问表的实用程序	177
17.4.1	-F 路径或文件名	177
17.4.2	-H 表头名	178
17.4.3	-L 每页行数	178
17.4.4	-O 输出文件名	178
17.4.5	-P 页宽	178
17.4.6	-S 包含拒选符号的文件名	178
17.4.7	-X 拒选符号的前导字符序列	178
第十八章	XAC 预处理器	178
18.1	XAC 预定义宏	179
18.2	#pragma 编译控制伪指令	179



第十九章 XAC 宏汇编器 .....	179
19.1 序 言 .....	181
19.2 XA 汇编源文件语句 .....	181
19.2.1 字符集 .....	181
19.2.2 数 .....	181
19.2.3 分隔符 .....	181
19.2.4 特殊字符 .....	181
19.2.5 标识符 .....	182
19.2.6 汇编生成的标识符 .....	182
19.2.7 位置计数器 .....	182
19.2.8 寄存器符号 .....	182
19.2.9 字符串 .....	182
19.2.10 暂时标号 .....	182
19.2.11 表达式 .....	182
19.2.12 汇编语句的格式 .....	183
19.3 XA 汇编伪指令 .....	183
19.3.1 伪指令语句格式 .....	183
19.3.2 PUBLIC .....	183
19.3.3 EXTRN .....	184
19.3.4 GLOBAL .....	184
19.3.5 END .....	184
19.3.6 程序段(PSECT) .....	184
19.3.7 ORG .....	185
19.3.8 EQU 和 SET .....	185
19.3.9 DB 和 DW .....	186
19.3.10 DF .....	186
19.3.11 DS .....	186
19.3.12 IF ELSE EKSEIF ENDIF .....	186
19.3.13 SIGNAT .....	186
19.3.14 控制选项伪指令行 .....	187
19.4 宏 .....	187
19.4.1 MACRO ENDM .....	187
19.4.2 LOCAL .....	188
19.4.3 REPT .....	188
19.4.4 IRP .....	189
19.4.5 IRPC .....	190
19.5 XA 汇编命令行 .....	190
19.5.1 XA 汇编命令行格式 .....	190

19.5.2 汇编选项	190
<b>第二十章 HLINK 连接器</b>	<b>191</b>
20.1 简介	192
20.2 连接与定位(或装载)的基本概念	192
20.3 连接命令	192
20.4 OBJTOHEX 实用程序	193
<b>第二十一章 HPDXX 51XX 集成开发平台</b>	<b>194</b>
21.1 安 装	196
21.1.1 MS_DOS 下的安装	196
21.1.2 UNIX 操作系统下的安装	196
21.2 快速入门	196
21.2.1 简单程序示例	197
21.2.2 使用 HPDXX	197
21.2.3 使用 XAC 命令行	197
21.2.4 运行程序	197
21.3 HPDXX 用户接口	198
21.3.1 监视器模式必性选择	198
21.3.2 菜单命令操作	198
21.4 HPDXX 菜单命令快览	199
21.4.1 系统子菜单(<<>>)	202
21.4.2 File 子菜单	202
21.4.3 Edit 子菜单	202
21.4.4 Option 子菜单	202
21.4.5 Compile 子菜单	203
21.4.6 Make 子菜单	203
21.4.7 Run 子菜单	204
21.4.8 Utility 子菜单	205
21.4.9 Help 子菜单	206
21.5 HPDXX 编辑器	207
21.6 编译连接一条龙示例	207
<b>附 录</b>	
<b>附录 A C51 函数库</b>	<b>209</b>
A.1 数学函数	209
A.1.1 函数名:abs, cabs, fabs, labs	209
A.1.2 函数名:exp, log, log10	210

A.1.3	函数名:sqrt	210
A.1.4	函数名:rand, srand	211
A.1.5	函数名:cos, sin, tan	211
A.1.6	函数名:acos, asin, atan, atan2	212
A.1.7	函数名:cosh, sinh, tanh,	212
A.1.8	函数名:fpsave, fprestore	213
A.1.9	函数名:ceil	214
A.1.10	函数名:floor	214
A.1.11	函数名:modf	214
A.1.12	函数名:pow	215
A.2	标准化 I/O 函数	215
A.2.1	函数名:_getkey( )	216
A.2.2	函数名:getchar	216
A.2.3	函数名:gets	216
A.2.4	函数名:ungetchar	217
A.2.5	函数名:_ungetkey	217
A.2.6	函数名:putchar	218
A.2.7	函数名:printf	218
A.2.8	函数名:sprintf	220
A.2.9	函数名:puts	221
A.2.10	函数名:scanf	221
A.2.11	函数名:sscanf	223
A.3	动态存储函数	223
A.3.1	函数名:calloc	224
A.3.2	函数名:free	224
A.3.3	函数名:int _mempool	225
A.3.4	函数名:malloc	225
A.3.5	函数名:realloc	226
A.4	字符归类函数	226
A.4.1	函数名:isalpha	226
A.4.2	函数名:isalnum	227
A.4.3	函数名:isctrl	227
A.4.4	函数名:isdigit	228
A.4.5	函数名:isgraph	228
A.4.6	函数名:isprint	229
A.4.7	函数名:ispunct	229
A.4.8	函数名:islower	230
A.4.9	函数名:isupper	230
A.4.10	函数名:isspace	231

A.4.11	函数名: isxdigit .....	231
A.4.12	函数名: toascii(参数宏) .....	232
A.4.13	函数名: toint .....	232
A.4.14	函数名: tolower .....	232
A.4.15	函数名: tolower(参数宏) .....	233
A.4.16	函数名: toupper .....	233
A.4.17	函数名: _toupper(参数宏) .....	234
A.5	字符串函数 .....	234
A.5.1	函数名: memchr .....	234
A.5.2	函数名: memcmp .....	235
A.5.3	函数名: memcpy .....	236
A.5.4	函数名: memccpy .....	236
A.5.5	函数名: memmove .....	237
A.5.6	函数名: memset .....	237
A.5.7	函数名: strcat .....	238
A.5.8	函数名: strncat .....	238
A.5.9	函数名: strcmp .....	239
A.5.10	函数名: strncmp .....	239
A.5.11	函数名: strcpy .....	240
A.5.12	函数名: strncpy .....	240
A.5.13	函数名: strlen .....	241
A.5.14	函数名: strchr, strpos .....	241
A.5.15	函数名: strrchr, strrpos .....	242
A.5.16	函数名: strspn, strcspn, strpbrk, strrpbrk .....	243
A.6	字符串转换函数 .....	244
A.6.1	函数名: atof .....	244
A.6.2	函数名: atol .....	245
A.6.3	函数名: atoi .....	245
A.7	变参数函数 .....	246
A.7.1	宏名: va_list .....	246
A.7.2	宏名: va_start(va_list ap, last_argument) .....	246
A.7.3	宏名: type va_arg(va_list ap, type) .....	246
A.7.4	宏名: va_end(va_list ap) .....	246
A.8	全程跳转函数 .....	248
A.8.1	函数名: setjmp .....	248
A.8.2	函数名: longjmp .....	248
A.9	内部函数 .....	250
A.9.1	函数名: _crol_, _irol_, _lrol_ .....	250
A.9.2	函数名: _cror_, _iror_, _lror_ .....	250

A.9.3 函数名: _nop_ .....	251
A.9.4 函数名: _testbit_ .....	251
A.10 抽象数组 .....	252
A.10.1 函数名: CBYTE, BBYTE, PBYTE, XBYTE .....	252
A.10.2 函数名: CWORD, DWORD, XWORD, PWORD .....	252
<b>附录 B C51 编译器使用错误提示</b> .....	253
B.1 前 言 .....	253
B.2 致命错误 .....	253
B.3 语法及语义错误 .....	255
<b>附录 C L51 连接/定位器使用错误提示</b> .....	267
C.1 前 言 .....	267
C.2 L51 警告 .....	267
C.3 L51 错误 .....	269
C.4 L51 致命错误 .....	272
C.5 例外信息 .....	275
<b>附录 D C51 的极限值</b> .....	276
<b>附录 E XAC 运行时间库函数</b> .....	277
E.1 ACOS .....	277
E.2 ASCTIME .....	277
E.3 ASIN .....	278
E.4 ASSERT .....	279
E.5 ATAN .....	280
E.6 ATOF .....	280
E.7 ATOI .....	281
E.8 ATOL .....	281
E.9 BSEARCH .....	282
E.10 CALLOC .....	283
E.11 CEIL .....	284
E.12 CGETS .....	284
E.13 COS .....	285
E.14 COSH, SINH, TANH .....	286
E.15 CPUTS .....	286
E.16 CTIME .....	287
E.17 DI, EI .....	287
E.18 DIV .....	288



E.19	EXIT	289
E.20	EXP	289
E.21	FABS	290
E.22	FLOOR	290
E.23	FREE	291
E.24	FREXP	292
E.25	GETC	292
E.26	GETCH, GETCHE, UNGETCH	293
E.27	GETS	293
E.28	GMTIME	294
E.29	ISALNUM, ISALPHA, ISDIGIT, ISLOWER 等	295
E.30	KBHIT	296
E.31	LDEXP	296
E.32	LDIV	297
E.33	LOCALTIME	298
E.34	LOG, LOG10	299
E.35	LONGJMP	299
E.36	MALLOC	300
E.37	MEMCHR	301
E.38	MEMCMP	302
E.39	MEMCPY	303
E.40	MEMMOV	303
E.41	MEMSET	304
E.42	PERSIST_CHECK, PERSIST_VALIDATE	304
E.43	POW	305
E.44	PRINTF, VPRINTF	306
E.45	PUTCH	308
E.46	PUTS	308
E.47	QSORT	309
E.48	RAND	310
E.49	REALLOC	310
E.50	SCANF, VSCANF	311
E.51	SET_VECTOR	312
E.52	SETJMP	313
E.53	SIN	314
E.54	SPRINTF, VSPRINTF	315
E.55	SQRT	315
E.56	SRAND	316
E.57	SSCANF, VSSCANF	317

E.58	STRCAT	318
E.59	STRCHR	318
E.60	STRCMP	319
E.61	STRCPY	320
E.62	STRLEN	320
E.63	STRNCAT	321
E.64	STRNCMP	322
E.65	STRNCPY	322
E.66	STRRCHR	323
E.67	TAN	324
E.68	TOLOWER, TOUPPER, TOASCII	324
E.69	VA_STSRT, VA_ARG, VA_END	325
附录 F	XAC 使用错误信息	327
附录 G	HTDXA 菜单命令热键	363

# 第一部分

# C 语言基础



## 第一章 词 法

语言用语句表达思想。语句由单词组成,所以先讲词法。

C 语言的单词有六种:

- 标识符(identifier)
- 关键字(keyword)
- 常 量(constant)
- 字符串(character string)
- 运算符(operator)
- 分隔符(punctuator 或 separator)

为了程序员能够自由地书写语句,允许单词之间的间隔距离任意(包括换行、续行),为此增加了空白符(nul)。

为了程序员能够对语句表述的内容加以注释,提高程序的可读性,又规定了注释符为 `/* */`。

下面分述之。

### 1.1 标识符

凡是在计算机中占有存储器位置的量都叫做对象。

对象分两大类:

- 数据
- 程序

数据是计算机要处理的目标物,而程序是关于处理行为的描述。

为了惟一地表达单个或成块的数据,还有单个或成块的程序语句,C 语言将它们每一个都用标识符命名,然后才能使用。

命名所用标识符的严格定义如下:

- 由英文字母 A~Z 和 a~z、阿拉伯数字 0~9 和下划线组成。
- 第一个字符一定是字母或下划线,其余字符可任意排列与组合。

- 最大长度因机器而异,一般为缺省值 32 个字符。
- 不应与关键字(见 1.2 节)相重。

## 1.2 关键字

关键字是 C 语言和 C 编译器专用的字符序列。选用标识符不可与关键字重名。

表 1.1 是 C 编译器的关键字。

表 1.1 关键字

_asm	else	_interruptpr	short
asm	enum	interruptpr	signed
auto	_es	_loadds	sigeof
break	_expert	_long	_ss
case	extern	long	static
_cdecl	_far	_near	struct
cdecl	far	near	sunteh
char	_fastcall	new	template
class	float	operator	this
const	for	_pascal	typedef
contincle	friend	pascal	union
_cs	goto	private	ungigned
defantr	_huge	protected	virtual
delele	huge	public	void
do	if	register	volatile
double	inline	return	while
_ds	int	_saveregs	
		_seg	

## 1.3 常 量

C 语言支持四种常量,即:

- 整数常量
- 浮点数常量
- 枚举量常量
- 字符常量

### 1.3.1 整数常量

整数常量就是数学中由负无穷大到正无穷大之间的某一确定的整数。整数在计算机中需要根据其值的大小分配不同的空间来存储。按照整数的取值范围,在 C 语言中分为:短整型数(short)、整型数(int)和长整型数(long)。它们包括正整数和负整数。如果只表示正整数,需用无符号整型数;无符号短整型数(unsigned short)、无符号整型数(unsignedint)和无符号长

整型数(unsigned long)。表 1.2 给出它们的数值范围,同时给出了十进制和十六进制表示的整数范围。C 语言用 0x 前缀表明后续数字为十六进制,以区别于十进制数。十进制数不需要加任何前缀或后缀。为了明显地表明某整数是长整型数,在数字后加 L(或 l),无符号数用 U(或 u)作后缀,如表 1.2 所示。

表 1.2 C 中整型常数及其整数范围

整数类型	取值范围		位 数 (二进制)
	十进制	十六进制	
短整型数 (short)	-128 ~ +127	0x800 ~ x7F	8
无符号短整型数 (unsigned short)	000 ~ 255	0x00 ~ 0xFF	8
整型数 (int)	-32 768 ~ +32 767	0x8000 ~ 0x7FFF	16
无符号整型数 (unsigned int)	00 000 ~ 65 535u	0x0000 ~ 0xFFFFu	16
长整型数 (long)	-2 147 483 648l ~ +2 147 483 647l	0x80000000l ~ 0x7FFFFFFFl	32
无符号长整型数 (unsigned long)	0 000 000 000 ~ 4 294 967 295ul	0x00000000 ~ 0xFFFFFFFF ul	32

### 1.3.2 浮点数常量

浮点数常量即数值为实型数的常量。实型数有整数部分和小数部分。根据实型数的数值范围的大小在计算机中用浮点数(float)、双精度浮点数(double)和长双精度浮点数(long double)表示。它们在计算机中分别占用二进制的 32 位、64 位和 80 位,按较为复杂的二进制阶码和尾数的格式存储。要想知道这个浮点数的十进制实数是多少需要经过转换。C 语言中专有库函数做这种复杂的转换。用户与计算机交互要用十进制的实数。所以,在计算机内部必须进行实数与浮点数的转换。表 1.3 中仅给出各种浮点数所能表达的十进制实型数数值范围。表中采用了只保留一位整数部分的科学指数表达法。

表 1.3 C 语言浮点数类型及取值范围

类 型	范 围	占用内存位数	注
单精度 (float)	$-3.4 \times 10^{-38} \sim 3.4 \times 10^{+38}$	32	精确到 7 位,用于科学计算
双精度 (double)	$-1.7 \times 10^{-308} \sim 1.7 \times 10^{+308}$	64	精确到 15 位,用于科学计算
长双精度 (long double)	$-3.4 \times 10^{-4932} \sim 1.1 \times 10^{+4932}$	80	精确到 19 位,用于财务



### 1.3.3 枚举常量

枚举常量是用关键字 `enum`(枚举类型)按下面格式说明的标识符:

#### 1. 原型说明

```
enum 枚举类标识符  
{ 枚举常量 1[ =n1][, 枚举常量 2[ =n2]]...};
```

#### 2. 定义性说明:

```
enum 枚举类标识符 枚举常量
```

其中:

(1) `n1, n2, ...` 是不可重复的、可不连续的、可取负值的整型数。

(2) 不用等号赋值的枚举常数取前一枚举常量的紧跟后续整数。

(3) 第一个枚举常量不赋初值时,取缺省值 0。

(4) 原型说明是给出枚举常量的一个样板,并未为样板分配内存。它的名字是枚举类标识符。

(5) 定义性说明按照原型分配内存,并起名为枚举常量。所以,枚举常量是可以进行存取的对象。

枚举类是 C 语言数据类型的一种,其目的在于给一有限集合的各元素赋以惟一的常量,以便对集合中各元素按数字进行处理。

[例 1.1] `enum days {sun, mon, tue, wed, thu, fri, sat};`

```
enum days payday; /* 定义 payday 为枚举类型 days */
```

```
payday = wed; /* 将周三的 3 赋给变量 payday 以便作为发薪日处理 */
```

因第一个枚举常量未赋初值,故取缺省值 0。后面的枚举量都未赋初值,取前面枚举常量的紧跟整数,故 `sun` 为 0, `mon` 为 1, `tue` 为 2, `wed` 为 3……。

### 1.3.4 字符常量

字符常量是由单引号括起的单个字符。它的值取本字符在本机器字符集中所对应的值,如 '0' 为 PC 机中所用 ASCII 字符集中的 0x31 或 IBM 巨型机所用的 EBCDIC 字符集中的 0x41。使用字符常量的好处是直观和便于在不同机器间实现软件的移植。最常用的字符集是 ASCII 字符集。

对于字符集中的非图形字符,如控制符,可以用转义字符表示。转义字符是反斜杠(\)加特定字符,如表 1.4 所示。

表 1.4 C 中非图形字符的转义字符常量

非图形字符	记号	转义序列	值	注
换行	NL(LF)	<code>\n</code>	0x0A	用有一定含义的字符加转义前缀 \ 表示
回车	CR	<code>\r</code>	0x0D	
水平制表	HT	<code>\t</code>	0x09	
垂直制表	VT	<code>\v</code>	0x0B	
退格	BS	<code>\b</code>	0x08	

续表 1.4

非图型字符	记 号	转义序列	值	注
走 纸	PF	\f	0x0C	
响 铃	BEL	\a	0x07	
反斜杠	\	\\	0x5C	因本字符已被专用, 其自身需加转义前 缀\表示
单引号	'	\'	0x27	
双引号	"	\"	0x22	
问号	?	\?	0x3F	

字符常量在计算机字符集中用 8 位无符号字符量或 8 位有符号的字符量表示, 它们的取值范围见表 1.5。

表 1.5 C 语言字符量

类 型	范 围		位 数	注
	十进制	十六进制		
无符号字符量	0~255	0x0~0xFF	8	ASCII 字符量和较少的数
有符号字符量	-128+127	0x80~0x7F	8	ASCII 字符量和较少的数

无符号和有符号字符量(unsigned char 和 char)与无符号和有符号短整型数(unsigned short 和 short) 都是 8 位二进制数, 实际上是一样的。今后, 只用 unsigned char 和 char, 而不用 unsigned short 和 short。

## 1.4 字符串

字符串是用双引号括起的字符序列。字符序列可以任意长, 其间可以包含转义序列、标点符号、空格等。

字符串也可以是空串, 即字符数为 0, 表示为“”。

[例 1.2] 普通字符串。

“This is a character string.”

[例 1.3] 带转义序列的字符串。

“\t This is a long \\ \n \t string example!”

相当于:

This is a long \  
string example!

字符串在内存中存放时, 在尾部自动加了一个串尾转义序列 \0, 作为结尾标志。所以, 字符串在内存中占有的字节数为字符数加一。

## 1.5 运算符

C 语言中, 运算符特别丰富, 如表 1.6 所示。按其在表达式中的作用, 可以分成算术运算

符、逻辑运算符、关系运算符、位逻辑运算符、赋值运算符等。按其在表达式中与操作数之间的关系可分为单目运算符、双目运算符和三目运算符。

运算符在表达式中的运算次序有先有后,运算次序根据运算符的优先级来确定,运算符的优先级见表 1.6 的第一列。优先级高的(数字大)先运算。运算符与操作数及同级运算符之间的结合关系,有从左到右结合和从右到左结合之别。见表 1.6 的最后一列。

表 1.6 运算符及其特性

优先级	运算符	功 用	适用类	结合性
15	( )	类型强制	参数表	从左到右
15	[ ]	数组下标	数组	从左到右
15	→	存取成员	结构	从左到右
15	*	存取成员	结构	从左到右
14	!	求逆	逻辑量	从右到左
14	~	取反	字位量	从右到左
14	++	加单位量	算术量	从右到左
14	--	减单位量	算术量	从右到左
14	-	取负	算术量	从右到左
14	&	取地址	指针	从右到左
14	*	取内容	间接取	从右到左
14	(类型名)	强制转换	类型转换	从右到左
14	sizeof	求类型单位量	类型量	从右到左
13	* / %	乘,除,取模	算术	从左到右
12	+ -	加,减	算术	从左到右
11	<< >>	左移位,右移位	字位量	从左到右
10	<	小于	关系	从左到右
10	<=	小于等于	关系	从左到右
10	>	大于	关系	从左到右
10	>=	大于等于	关系	从左到右
9	= =	恒等	关系	从左到右
9	! =	不等	关系	从左到右
8	&	字位与	字位量	从左到右
7	^	字位异或	字位量	从左到右
6		字位或	字位量	从左到右
5	&&	逻辑与	逻辑量	从左到右
4		逻辑或	逻辑量	从左到右

续表 1.6

优先级	运算符	功 用	适用类	结合性
3	? :	条件表达式	条 件	从右到左
2	* =	运算并赋值	算术量	从右到左
2	/ =	运算并赋值	算术量	从右到左
2	% =	运算并赋值	算术量	从右到左
2	+ =	运算并赋值	算术量	从右到左
2	- =	运算并赋值	算术量	从右到左
2	<< =	移位并赋值	字位量	从右到左
2	>> =	移位并赋值	字位量	从右到左
2	& =	字位与并赋值	字位量	从右到左
2	^ =	字位异或并赋值	字位量	从右到左
2	=	字位或并赋值	字位量	从右到左
2	=	赋 值	算术量	从右到左
1	,	按原规律依次向后运算	按序运算	从左到右

### 1.5.1 单目运算符

单目运算符(只要求一个运算数的运算符)如下所示:

单目运算符	操作数	单目运算符
*	表达式	
-	表达式	
!	表达式	
~	表达式	
&	表达式	
sizeof (类型名)	(表达式或类型名)	
++	左值	
	左值	++
--	左值	
	左值	--

\* , - , ! , & , sizeof(类型名)的操作数一定在单目运算符的右侧。++和--的操作数必须是左值,即它们必须是在内存中确实存在的对象,且为变量。左值可放在运算符的右侧,也可放在左侧,但意义不同。

(1) 单目运算符“\*”——所作用的操作数仅限于非空的指针。运算的结果为指针所指地址的内容。只要非空的指针不是常量指针,其操作结果可作为左值使用。

- (2) 单目运算符“ $-$ ”——求表达式的负值,即表达式结果值的 2 的补码。
- (3) 单目运算符“ $!$ ”——求表达式的逻辑非值,即将 0 变为非 0,非 0 变为 0。
- (4) 单目运算符“ $\sim$ ”——求表达式的反码或为 1 的补码,即将原值按位取反(1 变 0, 0 变 1)。
- (5) 单目运算符“ $\&$ ”——求表达式的地址。
- (6) 单目运算符“ $++$ ”——如果放在左值的左侧,是将左值加 1 个单位。如果放在左值的右侧,是将左值先记录下来之后,再将该左值加 1 个单位。单位是指该左值类型的字节数。
- (7) 单目运算符“ $--$ ”——与单目运算符“ $++$ ”相类似,不过不是加 1 而是减 1 个单位。
- (8) 单目运算符“ $\text{sizeof}$ ”(表达式或类型名)——求作为参数的表达式或类型名的类型字节数。
- (9) 单目运算符“(类型名)”——作用于其右侧的操作数。对其右侧的操作数进行类型的强制转换,转换为圆括号内指定的类型。

### 1.5.2 双目运算符

双目运算符又可分为算术运算符、关系运算符、位运算符和逻辑运算符。

#### 1. 算术运算符

算术运算符有  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$  五种。

·加法运算符“ $+$ ”——进行加法运算时,应保证左右两操作数的类型相同。如果类型不同,可以合法地转换为同一类型的将自动转换为同一类型,然后再相加;不能转换为同一类型的不得相加。

·减法运算符“ $-$ ”——运算规则与加法相似。

·乘法运算符“ $*$ ”——对左右两侧的操作数进行乘法运算。编译器对处于同一优先级上的连乘将自动重新优化排列,以求高效。

·除法运算符“ $/$ ”和求模运算符“ $\%$ ”——左侧操作数除以右侧操作数所得的商和余。误以零为除数求商和求模时,均产生溢出错误。两整型数的商一般并非整型数。两正整数相除,其商仍为正,其值向下取整数。两数有一个为负,其商为负,其值取整的方式与机器有关。一般情况下,余数应与被除数取同号,且余数应满足下列等式:

$$(a/b) * b + a \% b = a$$

故有:

商数	余数
$5/3=1$	$5\%3=2$
$6/3=2$	$6\%3=0$
$-5/3=-1$	$-5\%3=-2$
$5/(-3)=-1$	$5\%(-3)=2$

#### 2. 关系运算符

C 语言中的关系运算符有两组:

$>$   $>=$   $<$   $<=$  和  $=$   $!=$

·前组的优先级高于后组。各组的结合关系均为从左到右。

·在关系运算中,若关系成立,其结果为 1;不成立,为 0。0 和 1 均视为整型数,且可执行



通常的算术转换,但不看成是一般语言中的布尔量(TRUE 或 FALSE)。

·两个指针也可参与比较,结果取决于地址空间的前后位置。如果两指针指向同一数组的不同元素,就可以比较。

·关系运算符“==”是恒等关系,不是赋值。恒等关系若成立,其结果为 1;不成立为 0。关系运算符“!=”是不等关系。不等关系若成立,其结果为 1;不成立为 0。

### 3. 字位运算符

C 语言的字位运算符有四组:

<< 和 >> ——左移位和右移位。

& ——按位与。

^ ——按位异或(或按位加)。

| ——按位或。

·这四组运算符分属不同的优先级。

·它们的操作数必须是整型数。

·“<<”是左移位运算符,将其左侧操作数按右侧操作数指示的位数向左移位。运算的规则是:先将右侧操作数化为 int 类型,再进行左移位,移完之后又还原成原类型。因左移而在右侧空出来的位补 0。右操作数不应为负,且其值应小于左操作数的位数,否则结果不定。

·“>>”是右移位运算符,将其左侧操作数按右侧操作数指示的位数向右移位。右侧操作数应小于左侧操作数的位数。操作时先判左侧操作数是否为无符号数,是则将右移时左侧空出的位补 0;否则按有符号数处理,左侧空出的位填入与符号位相同的值。

[例 1.4] 有无符号整型数(unsigned int)X, 试从 X 的第 P 位开始提取低 n 位。有:

$$X = X >> (P+1-n) \& \sim(\sim 0 << n);$$

运算后 X 中为所求值。

### 4. 逻辑运算符

C 语言有两个以逻辑量为操作数的逻辑运算符:

&& ——逻辑与操作。

|| ——逻辑或操作。

·逻辑与的优先级高于逻辑或。

·“&&”和“||”两侧的操作数均被看做是逻辑量。在 C 中将非 0 看做逻辑 1(TRUE), 0 看做逻辑 0(FALSE)。

[例 1.5] X=1, Y=2, 则:

X && Y 结果为逻辑真,即 1(因 x, y 均不为 0)。而进行按位与时:

X & Y 结果为 0(注意:并非逻辑 0 (FALSE))。

·“&&”和“||”的操作规律为:只要有一操作数为逻辑 0,“&&”结果为 0,即 0 && X(任意值),结果为逻辑 0(FALSE);只要有一个操作数为逻辑 1,“||”结果为 1,即 1 || X(任意值),结果为逻辑 1(TRUE)。

### 1.5.3 三目运算符

C 语言有一个独特的三目运算符:

关系表达式 1 ? 表达式 2 : 表达式 3

先计算关系表达式 1, 如果其值非 0 (逻辑 1) 则取表达式 2 的值, 否则取表达式 3 的值。  
三目运算符在赋值语言中可作表达式使用。

[例 1.6]  $z = (a > b) ? a : b;$

它相当于条件语句:

```
if ( a > b )
    z = a;
else
    z = b;
```

其结果是将 a, b 中的大者赋于变量 z。

#### 1.5.4 赋值运算符

C 语言的赋值运算符分简单赋值运算符和复合赋值运算符。

·简单赋值运算符: =

·复合赋值运算符:

+	=	}	(算术运算符)
-	=		
*	=		
/	=		
%	=		
>>	=	}	(位运算符)
<<	=		
&	=		
^	=		
	=		

简单赋值运算符和复合赋值运算符的左侧, 必须是左值。复合赋值运算符实际上是对两侧的操作数先进行运算符指定的运算(如“+”= 中的“+”, “\*”= 中的“\*”), 之后, 再对左侧的左值赋值。

[例 1.7]  $X += Y + 1$  等价于  $X = X + (Y + 1)$

$X *= Y + 1$  等价于  $X = X * (Y + 1)$

复合赋值运算不仅书写简单, 而且有助于编译器产生高效率的代码。

#### 1.5.5 逗号运算符

下面是使用逗号运算符的一般格式:

表达式 1, 表达式 2, 表达式 3, ...

运算时是由左向右进行, 先运算表达式 1, 然后, 保留其类型再做后续表达式运算, 依次类推。

[例 1.8] 下述 for 语句中的初始化部分和增量部分使用了逗号运算符。

```
for(i=0, j=strlen(s); i < j; i++, j--)
```

[例 1.9] 下述函数调用语句实参表的第二参数使用了逗号运算符。

```
func(i, (j = 1, j + 4), k);
```

因实参表中的逗号被用作参数之间的分隔符, 为避免混淆将有逗号运算符的表达式用圆括号对括起, 上述调用语句相当于:

```
func(i, 5, k);
```

### 1.5.6 函数参数运算符

函数表达式格式为:

定义时      类型说明符    函数名(形式参数表)|...|

引用时      类型说明符    函数名(形式参数表);

调用时                      函数名(实际参数表);

在词法分析中被判明为实参时, 函数参数运算符“()”对各实参的类型有强制转换的作用, 转换的目标类型为函数定义时该实参所对应的形参的类型。

### 1.5.7 数组下标运算符

使用数组下标运算符的表达式如下:

数组名[下标表达式]

“[]”是从左到右结合的单目运算符, 与左侧的数组名相结合构成数组表达式。表达式的结果是由下标所指定数组元素的内容:

\* (数组名 + 下标表达式 \* sizeof (类型说明符))

其中, 数组名和下标表达式二者必须有一个是指针类型而另一个为整型。

### 1.5.8 结构/联合成员运算符

使用结构/联合成员运算符的表达式格式为:

结构名·标识符

联合名·标识符

成员运算符为双目运算符, 左侧应是内存中实有的结构对象名或联合对象名。换句话说, 成员运算符的左操作数应为左值。运算符右操作数应为成员名。表达式运算结果是成员的内容。

### 1.5.9 结构/联合指针运算符

使用结构/联合指针运算符的表达式格式为:

结构指针→标识符

联合指针→标识符

本指针运算符为双目运算符, 左侧操作数必须是结构型指针或联合型的指针, 右侧操作数为成员名。表达式的结果是结构成员或联合成员的内容。

## 1.6 分隔符

C 语言中使用的分隔符有:

[ ] ( ) { } , ; : ... \* = #

分隔符中有许多是与运算符相重的, C 语言中允许这种重用。要想区别它们, 需看它们在上下文中的位置和所起的作用。用在表达式中的上列分隔符一定是运算符, 只有一个例外——圆括号。它在复杂的数学表达式中作为隔离符将圆括号之间的表达式部分先行运算, 然后将其结果再参与其余复杂表达式的常规优先级运算。这样的圆括号并非运算符, 而是分隔符。

初学 C 语言的读者请先放弃本节的下述部分。因为它需要后续的知识方好理解, 况且, 这部分对初学的编程人员并不重要。当对 C 有了一定的认识之后, 再阅读这部分内容。

### 1.6.1 方括号分隔符

方括号分隔符用于数组变量的说明。字符数组变量的定义说明格式如下:

```
char str[] = "string";
```

“[]”不是运算符, 作为分隔符说明其左侧的标识符是字符数组变量, 并且方括号中间缺少下标变量是可以的。因为编译器将根据“=”号后的字符串长度再加 1 个字节的‘\0’(字符串结尾符 nul) 安排内存的大小, 并将字符串复制到字符数组中。

二维数组变量定义说明格式如下:

```
int matrix[3][7];
```

“[]”也不是运算符, 不能按照表达式中的运算符对待。在这里, 括号中的数字不是为了计算元素位置的下标, 而是标明为本数组应分配内存的大小。在本例中应使用下式计算分配内存的大小(单位为字节):

```
3 * 7 * sizeof( int )
```

与计算元素的位置是两回事。

### 1.6.2 圆括号分隔符

```
d = c * (a + b);
```

用“()”隔离出  $a + b$ , 用以破例提前运算。

```
if (a == b) x++;
```

用“()”分离 if 所要求的关系表达式。

```
int (*fptr)();
```

用“()”分离出  $*fptr$ , 其中  $fptr$  是无参数并返回 `int` 类型的函数指针。本语句是关于函数指针的定义性说明。

### 1.6.3 花括号分隔符

```
if ( d == z )
```

```
{ ++x;
```

```
func();
```

```
}
```

用“{}”分离出多语句组成的复合语句, 作为 if 要求的语句使用。

### 1.6.4 逗号分隔符

```
void func (int n, float f, char ch);
```

逗号分隔符用在参数表中分隔各个参数。这里的逗号并非逗号运算符。

在函数调用中,如:

```
func((表达式 1, 表达式 2), f, ch );
```

第一个参数(表达式 1, 表达式 2)用了逗号表达式,其中的逗号是逗号运算符。而在参数表中的其他逗号为分隔符。

### 1.6.5 分号分隔符

分号分隔符用在语句的最后,作为语句终结符,如:

```
++a;
```

表达式为 0 终结符构成表达式语句,如:

```
for(i=0; i<w; i++)
```

```
{
```

```
;
```

```
}
```

在 for( ) 中的表达式语句用分号分隔符作为各语句的终结符:

```
i=0;
```

```
i<w;
```

在 for 的主体部分 { } 中用分号分隔符构成空语句。

又如,在函数的主体部分 { } 中用分号分隔符构成空函数如下:

```
void function()
```

```
{
```

```
;
```

```
}
```

### 1.6.6 冒号分隔符

在语句行中用冒号分隔符分隔出一个标识符,作为其后语句的标号。

```
start:
```

```
x=0;
```

```
...
```

```
goto start;
```

标号在 C 中是其后第一条语句的地址。它为 goto 语言设定控制的转入点。

### 1.6.7 省略号分隔符

在函数的参数表中,用省略号“...”作为分隔符,说明在它出现的地方开始,有可变个数的参数,如:

```
int printf(char *fmt, ...)
```





fmt 之后有可变的参数。具体是多少,由调用时实参表中所含变量的个数来决定。

### 1.6.8 星号分隔符

星号分隔符作为指针的说明符:

```
char *cptr;
```

分隔符 \* 加在标识符前,说明该标识符为指针名。指针的类型是用它指向内容的类型来定义的。本例中指针 cptr 的类型是 char。

标识符前加双星号,说明该标识符是二重指针:

```
char **cptr;
```

即它的内容是一重指针的地址。

### 1.6.9 等号分隔符

等号分隔符是作为初始化分隔符使用的。它将变量与变量的初值部分分隔出来。初值部分是为变量赋初值的,如:

```
char array[5] = {1,2,3,4,5};
```

```
int x = 5;
```

分别为数组的各元素并给变量赋初值。

### 1.6.10 预处理器伪指令符

如在程序的某行上,第一个非空白字符为“#”时,则该行是预处理器的伪指令语句行。它虽然写在源程序中,但并不产生程序代码,只是告知预处理器应如何操作,如:

```
# include <stdio.h>
```

是预处理器的包含伪指令语句。

## 1.7 空白符(nul)

用 C 语言书写源程序语句时,采用的是自由格式,如单词与单词之间可以随意空一个或一个以上的空白字符,也可空一个或一个以上的空行。空白符在 ASCII 字符集中用 nul 标注。它的字符值为零,用转义序列表示为 \0。水平制表符的转义序列为 \t、垂直制表符为 \r、换行符为 \n。注释符 /\* \*/ 所限定的字符区间虽然可以放作为注释的字符串,但是对于 C 语言来说都作为空白符处理。因为它们也都不是单词的有效成份。

因页面有限,语句并未结束而需要换入下个逻辑行续写时,在 C 语言中用反斜杠作为续行符,转到页面的下行继续输入本语句的后续内容。

## 第二章 说 明



计算机为高速处理外部世界的事务而存在。计算机在处理事务时，必须要有程序。程序由两部分内容所组成：一部分代表被处理的对象；另一部分是处理这些对象的语句。本章介绍对象的说明，第三章介绍语句。实际上，程序是对象说明和语句的集合。简单的程序放在一个源文件中，复杂的程序可以分放在多个源文件中，然后分文件进行编译，最后再连接起来成为一个可再定位的整体程序，经定位后，便是可执行的程序放在所谓的可执行目标文件中。

下面介绍本书在格式说明中的惯用符号：

[ ]——括号内的内容是可有可无的选项。括号内若有多个选项时，一次只能取一项或一项也不取。

| |——括号内的项是必须的。括号内有多项时，一次只能取一项，且必须取一项。

… ——重复前面的内容。

### 2.1 对象与左值

#### 2.1.1 对 象

在计算机内存中，被标识符说明过的区域就是对象。标识符代表这个对象的内容。使用经说明后的标识符就可以存取内存中这个区域的对象。

可被说明的对象有：

- 常量
- 变量
- 复合变量(数组、结构、联合、枚举常量)
- 函数

常量是程序执行过程中自始至终不变的量。它可以不占内存，也可以占用内存。如果不占用内存，就不必说明为对象。

常量中惟一例外的是枚举常量，枚举常量必须要说明为对象。所以，枚举常量必然是占用内存的对象。枚举常量的说明，在第一章已讨论过，不再重述。

实际上，说明为对象的常量是变量的特例。在说明变量时加上常量修饰符(const)予以限定即可(见后)。

如果常量不说明为对象，按照C语言的习惯应该用预处理器伪指令定义为宏。用宏名代替该常量在程序中使用。用宏名来表达常量是为了日后维护的方便。

C语言中的说明有两类：一类是定义性的说明，编译器遇到定义性说明要为其分配内存；另一类是引用性的说明，编译器不再为其分配内存。

引用性说明在多文件的程序中是必不可少的。变量在某个源文件中作过定义性说明，而在其他文件中想要引用，就得先作引用性说明，然后才可以使用。引用性说明的目的是表明这

个变量已在它处作过定义性说明,这里仅是引用而已,不需再重复分配内存。

允许在不同的源文件中用相同的标识符进行变量的定义性说明。这时,它们在内存中是不同的对象。但是,它们之间发生了重名。这和人类社会中两个不同的自然人重名一样。实际上,重名是难于避免的。重名会给管理带来混乱。重名的混淆毫无疑问应该避免,这也正是 C 要使用说明来解决的问题之一。解决的办法见后。

### 2.1.2 左 值

左值是可以放在赋值运算符左边的对象。因而,它必须既是对象,又要可变。

用上述条件来衡量不可作为左值的对象有:

- 常量
- 复合变量本身(数组、结构、联合、枚举常量)
- 函数本身

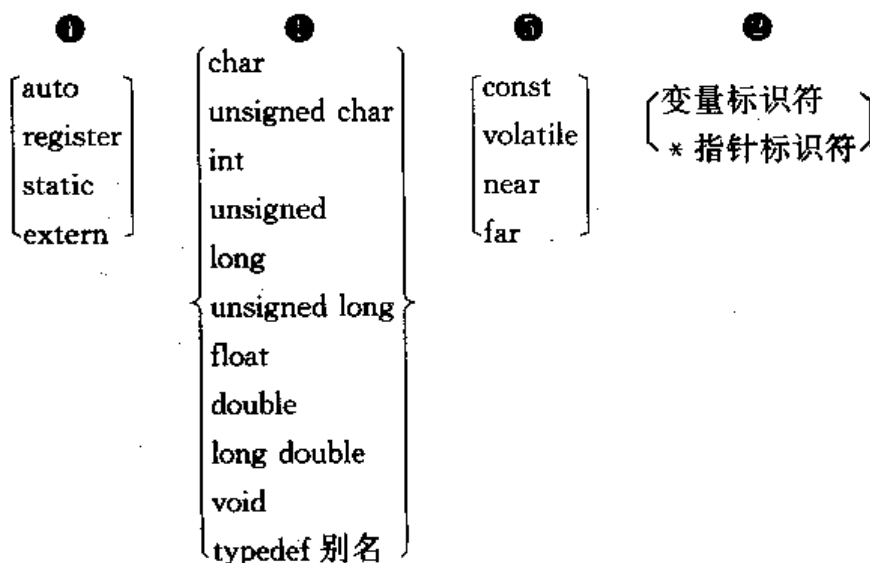
可以成为左值的对象有:

- 变量
- 复合变量的非常量分量(非常量的数组元素、结构和联合的非常量成员)
- 复合变量的指针、函数的指针、变量的指针

## 2.2 变量说明

格式如下:

[存储类说明符①] 类型说明符① [修饰符⑤] 标识符② [=初值③][, 变量标识符 [=初值③]]…;



其中:

(1) 类型说明符①——除以前介绍过的数学与字符等类型 9 种以外,还有 void 和由 typedef 定义的类型别名。void 将在稍后介绍。关于 typedef, C 语言允许用户为 C 语言固有的类型用 typedef 起别名。起别名的目的是为了增加程序可读性和移植程序时的方便。

定义类型别名的格式如下:

typedef C固有的简单类型或复合类型 别名标识符;

别名可以代替原来的类型,在说明中用做类型说明符。别名一般用大写字母以与原来的类型说明符相区别。

[例 2.1] typedef long BIGGY

BIGGY salary = 80000L;

(2) 标识符②——无“\*”为前导的标识符是变量标识符。变量标识符简称为变量。编译器为变量自动分配内存。

有“\*”为前导的标识符是指针标识符。注意,指针标识符是不带“\*”的标识符部分。指针标识符简称为指针。编译器为指针自动分配内存。指针的内容必须是地址。注意,指针说明中的类型说明符并非指针本身的类型,而是指针指向的对象的类型(指针的存储类和修饰符也是关于指针所指向对象的限定内容)。一般所说,指针是变量,即指针指向的对象是变量。

变量指针的类型可以是 void(即: void \* 标识符)。void 是抽象类型。抽象型指针所指对象的类型,可以是类型说明符中的任一类,在具体化时用类型强制来指定。

上述的类型说明符①和变量标识符②是说明中不可缺少的部分。下面所述的③ ④ ⑤是说明中可有可无的。

(3) 变量初始化部分③——初始化部分是可有可无的。

若不初始化, auto 存储类和 register 存储类的变量为随机值; static 存储类的变量被编译器自动清为 0; 对于指针, 无论什么存储类, 只要不给初值, 一律置为空指针(nul)。

如果需要初始化, 使用等号为先导的初值。数学类变量用整型数或浮点数作为初值。字符类用整型数或字符在字符集中对应的编码值作为初值。字符在字符集中对应的编码值在 C 中可用单引号对括起的字符代替。指针必须用地址量作为初值, 如 & 变量名等。对于 static 存储类初值只在定义说明时赋值一次。

(4) 存储类说明符④——指定被说明对象所在内存区域的属性。

·auto(自动存储类)——指定被说明的对象是放在内存的堆栈中的。C 语言把函数内说明的内部变量和指针, 还有函数的实参都放在堆栈中。它们随着函数的进入而建立, 随着函数的退出而自动地被放弃。在函数中说明的内部变量, 凡未加其他存储类说明的变量都是自动存储类。每次函数被调用时都是重新在堆栈中分配, 位置一般是不一样的。

如果函数要求内部变量的地址每次调用时是固定的, 它就不应放在堆栈中, 而应放在内存中。这时要用 static 存储类(见后)。

·register(寄存器存储类)——指定将变量放在 CPU 的寄存器中, 以求处理的高速。具体编译时不可用的寄存器, 将被分配到由存储模式决定的缺省存储类空间。

·extern(外部存储类)——在 C 语言中, 凡在函数内部作定义性说明的变量都是内部变量, 如前述的 auto 和 register 存储类变量。内部变量只在函数内部存在且可见(从而能够存取)。

在函数外部作定义性说明的变量都是外部变量。外部变量也叫做全局变量。外部变量是全程范围内存在的, 但能否可见被存取要看该外部变量定义性说明与使用它的函数之间的位置关系而定。如果是单文件的程序, 使用该外部变量的函数在外部变量定义性说明之后, 则可直接存取; 否则, 应在函数使用该外部变量之前加引用性说明。如果是多文件的程序, 在非该变量的定义文件中要使用它时, 必须先加引用性说明, 然后才能使用。

变量的引用性说明格式如下:

`extern 类型说明符 [修饰符] 变量名 [, 变量名]...;`

因为引用性说明不分配内存,所以不能有初值部分。

·static(静态存储类)——有内部静态和外部静态两种存储类。在函数内部,希望所定义的变量在离开这个函数到下次调用之间其值保持不变,这时就要加静态存储类说明。这是内部静态变量。内部静态变量在程序全程内存在,但只在本函数内可见(可存取)。在本函数外即不可见(从而不能存取它)。使变量在外部不可见,除去对这个内部的变量起保护作用之外,还给外部命名其他变量以自由,即使重名也不会混淆。在函数外部定义变量并说明为静态存储类的是外部静态变量。外部静态变量在程序全程内存在,但在定义它的范围以外,也是隐蔽起来的,使之不可见。对于只有一个源文件的程序来说,如果在文件开始定义为外部变量的加与不加 static 是一样的,也就是说,这个外部变量一定是全局变量。

(5) 修饰符①——修饰符用于对变量进行特殊地修饰。它不是必须的。

·const(常量修饰符)——指示被修饰的变量或变量指针是常量。在 C 语言中,把被 const 修饰过的量,在内存中开辟的一个常量区进行存放。任何程序对常量区域进行修改都是非法的。变量被 const 修饰后就不能再变了。这是使常量成为对象的唯一方法。这样做的好处是,给常量带上类型的属性。

对于指针有两种修饰的方法,它们的意义是不同的,如:

`int const * ptr; /* 说明指针指向的对象是常量。 */`

`int * const ptr; /* 说明指针本身是常量。 */`

·volatile(易失性修饰符)——指示所说明的变量或指针,是可以被多种原因所修改的。为此,禁止把它作为寄存器变量处理,也禁止对它进行任何形式的优化。譬如有的变量在中断服务程序中会被修改,有的会被 I/O 口修改,这种修改带有随机性,防止丢失任何一次这种修改,故要把它修饰为易失性的变量。

·near, far(近,远修饰符)——用于指示访问内存中变量在位置上的远近。

(6) 格式中“[, 变量标识符 [= 初值']]...;”的部分——说明在一个说明语句中可以同时说明多个同类型的变量。这些变量之间要用逗号隔开,每个变量是否赋初值是任选的。

(7) 说明语句一定要用终止符(;)结束。

## 2.3 数组说明

同类型变量的有序集合叫数组。

说明格式如下:

`[存储类说明符①] 类型说明符② [修饰符③] 标识符④ [= 初值符⑤][, 标识符 [= 初值]]⑥...;`





①	②	③
<div style="border: 1px solid black; padding: 5px; display: inline-block;">           auto register static extern         </div>	<div style="border: 1px solid black; padding: 5px; display: inline-block;">           char unsigned char int unsigned long unsigned long float double long double void typedef 别名         </div>	<div style="border: 1px solid black; padding: 5px; display: inline-block;">           const volatile near far         </div>

数组说明格式中的①, ①~③与2.2节的内容完全一样。需要阐述的是②和③项的内容。它们比较复杂, 为简明下面将①, ②和③栏一并讨论。

### 2.3.1 一维数组, 一维指针数组, 一维数组指针

#### 1. 一维数组

格式:

类型说明符① 标识符[常量表达式][={初值, 初值, ...}④];

例外 char① 标识符[ ]②="字符串";

标识符后跟单目运算符方括号是标识一个一维数组的特征。方括号在用于数组时, 其括号内必须放有常量表达式或是可以使用 sizeof 运算符和四则运算符等能比较简单地化为数学常量的表达式。初始化可有可无, 没有初始化时数组各元素为随机值; 需初始化时, 应使用等号为前导的花括号, 其中是用逗号分隔的初值表。初值的个数不允许多于常量表达式的值, 初值为0的元素可以只用逗号占位而不写初值。初始化表的项数少于常量表达式值是允许的, 其后的缺项由编译器自动以0值补足之。

字符型(char)数组是例外, 允许标识符后跟的方括号内为空, 但必须存在初始化部分。初始化部分是等号为前导的双引号括起的字符串。字符数组的长度由编译器自动完成, 其值为串中字符数加1(多结尾符'\0')。

#### 2. 数组的复合标识符(一维指针数组和一维数组指针)

类型说明符① \*标识符[常量表达式]②[={地址, 地址, ...}④];

类型说明符① (\*标识符)[ ]②[=数组标识符④];

前者是以指针为元素的数组, 所以是一维指针数组; 后者是一维数组的指针或叫数组指针。数组指针不需要指明所指向的数组有多少元素, 所以下标方括号可以是空的。

C语言用数组标识符作为所标识数组的首地址, 即数组第一个元素的地址。故应注意, 数组标识符并不是整个数组。需要指出, 指针和地址是有区别的。指针是个对象, 在内存中被分配有空间的。只不过该对象的内容不是变量, 而是可变的地址。而数组标识符只不过是已分配过内存的数组的首地址。也就是说数组标识符, 它并非是数组对象。

[例2.2] 用指针形式求数组元素。





```
int a[20], x;
int * p;
p = a;
或 p = &a[0]; 有 x = *(p + 3);
```

[例 2.3] 用数组形式求数组元素。

```
int a[20], x;
x = a[3];
```

### 2.3.2 多维数组, 多维指针数组, 多维数组指针, 多重指针

下面是关于二维数组的定义性说明格式(仅给出简化格式):

$$\begin{array}{c} \text{m} \\ \text{——} \\ \text{n} \quad \text{n} \\ \text{——} \quad \text{——} \end{array}$$

类型说明符① 标识符 [m] [n]② [= | {初值表} | {初值表}, ..., {初值表} | ③];

$$\begin{array}{c} \text{m} \\ \text{——} \\ \text{n} \quad \text{n} \\ \text{——} \quad \text{——} \end{array}$$

类型说明符④ \* 标识符 [m] [n]⑤ [= | {地址表} | {地址表}, ..., {地址表} | ⑥];

类型说明符① (\* 标识符)[] [n]② [= 数组标识符③];

类型说明符① \*\* 标识符② [= & 指针③];

1. 二维数组——格式中第一个说明是  $m \times n$  个元素的二维数组。当标识符独立使用时,它是二维数组在内存中的首地址。它与 & 标识符[0][0] 相同,但书写简单。数组标识符是地址常量,不能用作左值,可用在表达式中,而不可用于赋值运算符的左侧。在 C 语言中,二维数组在内存中的排列次序是先放第一行的  $n$  列,再放第二行的  $n$  列,直到第  $m$  行的  $n$  列。所以,说明中的初值也应按同样的顺序排列。初始化时,可以用省略某些元素初值的格式,但被省略元素的初值隐含为 0。

2. 二维指针数组、二重指针——格式中第二个说明是  $m \times n$  个元素的二维指针数组。标识符单独使用时,它代表的是指针数组的首地址。由于这个地址的内容是指针,再指向的内容才是变量。所以,指针数组标识符具有二重指针性质。因它是地址常量,不可以用为左值。但可以为二重指针赋初值,如:

```
int * addadd[3][4], b[5];
int ** ptr1, * ptr2;
ptr1 = addadd;
ptr2 = &b;
```

3. 二维数组指针——第三个说明是一个二维数组的指针,其标识符单独使用时是(一重)指针变量。为了使这个指针能指向二维数组的任一个元素,说明时,应给出第二个方括号中的下标常量,至于第一个方括号中的下标量无需给出,给出也不为错。

## 2.4 结构说明

不同类型变量的集合叫结构。

结构说明有原型法和类型别名法。

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

### 2.4.1 原型法

1. 两步法——先说明原型,再用原型做样板给出结构的定义性说明。

(1) 结构原型说明

```
struct 结构原型名
{ 类型说明符标识符[, 标识符 ...];
  [ 类型说明符标识符[, 标识符 ... ];
  ...
}
```

(2) 结构定义性说明

[存储类说明符] struct 结构原型名 标识符 [= {初值表} [, 标识符 [= {初值表}] ... ];

```
{static}           {结构名
extern}             { * 结构指针名}
```

2. 一步法

结构定义性说明

[存储类说明符] struct [结构原型名]①

```
{static}           |
extern}             | 类型说明符标识符[, 标识符 ...];
                    | [ 类型说明符标识符[, 标识符 ... ];
                    | ...
                    | } 标识符 [= {初值表} [, 标识符 [= {初值表}] ... ];
                    | {结构名
                    | { * 结构指针名};
```

一步法是两步法的合二为一。但它不够灵活,比如,只适合一次性做结构的定义性说明。在一步法定义性说明中,允许省略 [结构原型名]①。如果省略了原型名,则此结构不应作为函数的参数使用。

### 2.4.2 类型别名法

先为结构原型名起别名,再用别名做定义说明。

1. 给结构的原型起别名

```
typedef struct[结构原型名]②
{ 类型说明符 标识符 \[, 标识符 ...;
  [ 类型说明符 标识符 \[, 标识符 ...;
  ...
}
```

| 结构别名

2. 结构定义性说明

[存储类说明符] 结构别名 标识符 [= {初值表} [, 标识符 [= {初值表}] ... ];

(static extern)	(结构名 * 结构指针名)
--------------------	------------------

其中:结构别名●习惯上用大写字符。

[结构原型名]●可用可不用;习惯上不用。因为,一般说来,别名更具特色。

#### [例 2.4] 原型法

原型说明:

```
struct person
{
    char   name[16];
    char   address[32];
    char   sex;
    double salary;
};
```

定义性说明:

```
struct person worker[100], * pstruct;
```

#### [例 2.5] 类型别名法

定义别名:

```
typedef struct
{
    int day;
    int month;
    int year;
    int yearday;
    char mon - name[4];
} DAY;
```

定义性说明:

```
DAY birthday, holyday[10];
```

### 2.4.3 关于结构的讨论

1. 结构由各种数据类型的成员组成。成员之间没有次序关系,访问成员不按次序,而用结构成员名。

2. 成员可以是各种简单变量类型和复合变量类型,包括本结构的指针和位段结构(见后),惟独不可包括本结构自己。

3. 只有在定义性说明时,才可以整体性地为结构赋初值。在程序中,不能用语句整体性地给结构赋值。但可以对成员个别地进行赋值和取存操作。

4. 存取成员的方法有两种:

结构名·成员名

结构指针名->成员名

前者是结构首地址加偏移法,后者是指针值加偏移法。只要结构指针指在结构的首地址上,二者访问同一成员。

5. 对结构只能进行两种运算:一是前面说的对结构成员的访问;二是取结构的地址(& 结构名)。

超星浏览器提醒:  
使用本复制品  
请尊重相关知识产权!

6. 结构的成员可以是数组,数组的元素也可以是结构。即结构和数组可以互为嵌套。

#### 2.4.4 位段结构

位段结构是把 16 位的整型变量按位段定义成结构。

位段结构的说明格式如下:

```
struct [位段结构原型名] ②
{ 整数类型说明符 [位段名] ③ :位宽;
  [整数类型说明符 [位段名] ③ :位宽;
  ... ]
  标识符① [= {初值, 初值, ...}]④ [, 位结构名 [= {初值, 初值, ...}...⑤];
```

其中:

- (1) 标识符①部分是必有的。标识符是位段结构名。
- (2) [位段结构原型名]②部分是可有可无的。没有时,虽简单但灵活性较差。
- (3) [位段名]③部分是可有可无的。没有时,表示该位段未用或有用的位段一定得起名。
- (4) [= {初值, 初值, ...}]④部分是可有可无的。赋初值时,由低位开始,各位段用逗号分割。初值应按位段宽根据类型给适宜的符号数或无符号数。类型说明符可以是: char, unsigned char, int, unsigned。位段宽的取值范围为:1~16。
- (5) 可同时定义多个位段结构,但应用逗号分割。
- (6) 位段结构只能用于结构和联合中。

[例 2.6] struct mystruct

```
{ int i: 2;
  unsigned j: 5;
  int 3; /* 未用 */
  int k: 1;
  unsigned m: 5;
  a = {1, 2, 0, 7};
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
m					K	未用			j				i		
0~31					0~-1			0~31				1~-2			

## 2.5 联合说明

联合是在内存中定义的一段多种类型数据所共享的空间,空间的大小以最长的类型为准。联合说明的格式与结构完全相同。为精简,下面只给出一步法的定义性说明。联合说明的格式如下:

```
[存储类说明符] union [联合原型名] ①
{ (static) | 类型说明符 标识符[, 标识符 ... ];
  (extern) [ 类型说明符 标识符[, 标识符 ... ];
```

```

... ]
| 标识符 [= {初值表} [ , 标识符 [= {初值表} ] ... ] ;
{ 联合名
  * 联合指针名
}

```



[例 2.7] union myunion

```

{ int i;
  double d;
  char c;
} mu, *mup = &mu;

```

讨论：

(1) 联合与结构的说明格式一样，上面的格式只说明了一种。其他从略。

(2) 联合的操作与结构一样，只有成员存取和取地址操作：

存取：联合·成员名

联合指针→成员名

取地址：& 联合名

(3) 为联合的某成员赋值，其他成员也有所表现，但是一般是曲解的。只有对被赋过值的成员作存取操作才是正确的。因此，以向联合的成员赋值操作为界限，每轮从对成员赋值开始到下次对它成员赋值为止，只对赋值的成员进行取存才肯定是正确的。但是，这也并不是说非赋值的成员一定是不能用的。

(4) 联合可以用于数组和结构之中，数组和结构也可用于联合之中。

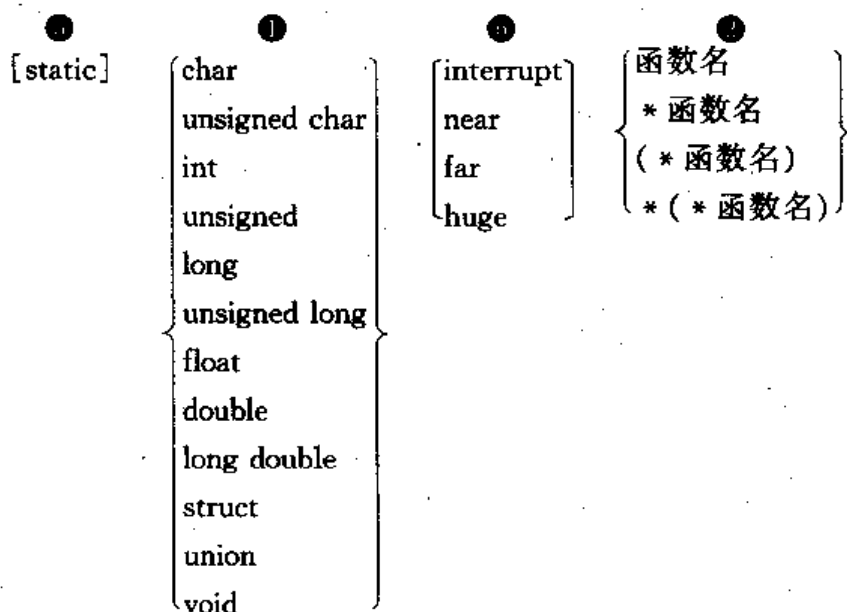
(5) 联合不能作为向函数传递的参数，也不能从函数返回联合。但是联合的指针可向函数传入或被返回。

## 2.6 函数说明

函数是 C 语言的核心。C 语言编写的程序由一个主函数和若干函数为骨架所组成。函数有定义性说明和原型说明。

### 1. 定义性说明格式

[存储类说明符] ⑤ 类型说明符 ① [修饰符] ② 标识符 ③ (参数表 ④) | 函数体 | ①



## 2. 原型说明格式

extern ⑤ 类型说明符 ① [修饰符] ③ 标识符 ② (参数表 ④);

其中: ① ② ③ ④ 项是函数定义性说明中必不可少的部分。可以在程序源文件的任意处进行函数的定义性说明, 并被分配内存。

· 类型说明符 ① 部分——用以说明函数返回值的类型。有简单类型(char, unsigned char, int, unsigned, long, unsigned long, float, double, long double)、复合类型(struct, union)和 void (无类型)。为说明函数返回的是指针, 在函数名前加“\*” (见标识符 ② 的说明部分)。

· 标识符 ② 部分——用以说明函数名。但函数名前加分割符“\*”时, 说明返回值是指针。然而再被圆括号括起如(\* 函数名), 则标识符是函数指针。如果括号外再加分割符“\*”, 如\*(\* 函数名), 则标识符是函数指针, 且函数的返回指针类型。

· (参数表 ④)——传入函数的形式参数表。形式参数表格式如下:

(类型说明符 变量名 [[, 类型说明符 变量名]...])

或 (void)

或 ( )

其中(void)说明无参数传入。有的 C 语言允许用空格代替 void 做为参数表, 即( )。

· {函数体} ⑤ 部分——函数体由复合语句构成, 详见第三章复合语句一节。

· [存储类说明符] ③ 部分:

**extern**——C 语言的函数都是全程序存在的, 在不加任何存储类说明的情况下都是全程序可见的。但是, 程序为多源文件时, 非定义函数的文件要调用该函数时, 需加原型说明。另外, 即使在定义函数的源文件中, 如果在函数定义之前超前调用, 也需要加原型说明。原型说明中必须加存储类说明符 extern。加原型说明的目的是配合编译器核对函数调用是否匹配。核对的项目包括函数名(大小写敏感)、参数的个数和类型、还有返回值类型。

**static**——为了提高函数的安全性, 在进行函数的定义性说明时, 可加 static 存储类说明符, 使函数对本文件的本定义之前的部分和对非定义文件隐蔽起来不会被调用。隐蔽的范围是静态定义以外的各文件和本文件静态定义之前的部分。

· [修饰符] ③ 部分——对函数起修饰作用。修饰符有:



interrupt——最重要的修饰符。它将函数修饰为中断函数。中断函数的最大特点是返回类型和参数均必须为 void。

函数经过 interrupt 修饰后,程序员只写中断服务程序的主体部分,中断服务程序中的保护现场前缀段和恢复现场的后缀段,均由编译程序完成。另外,编译程序还将 `ret` 指令改成 `reti` 指令。

near, far, huge——是用以规定函数的地址类型。它将复盖存储模式规定的函数缺省地址类型。它指明函数和被调用函数之间的距离的远近。`near` 为近调用(16 位段内地址)。`far` 为远调用(32 位段间地址)。`huge` 为规范化远调用(32 位段间规范地址)。

·关于函数原型说明格式——在同一文件之内,并在函数定义之后被调用者除外,凡在同文件之内超前调用的,或在非定义的其他文件中欲调用者,在调用前都必须先做函数的原型说明,然后才能调用。原型说明格式的特点是:定义说明中的①,②,③三部分必须存在,并在最前加 `extern`,在最后加终结符(;)构成。

## 第三章 语 句



### 3.1 语 句

C语言中的语句格式如下：

[标号:] 语句[;]

其中：

(1) 标号部分是可有可无的。一般在源文件中存在跳转语句时，才给跳转到的语句加标号部分，以指明转入的地址。标号部分由有效标识符后跟冒号分隔符组成。标号在本层复合语句中不应重名。

(2) C中的语句只有 10 种：

表达式语句	复合语句	条件语句	循环语句	开关语句
中断语句	接续语句	返回语句	跳转语句	空语句

(3) 语句结束部分一般要用分号做结束符。但有的语句自身有明确的间隔符时，分号可以省略。即所有的语句必需用间隔符结尾，但不一定是分号。

下面先介绍表达式，因为它是表达式语句的基础。然后，逐句介绍语句部分，介绍时标号不再列出。

### 3.2 表达式

C语言中的初等表达式有：

常量	字符串
变量	(表达式)
左值·标识符	表达式→标识符
数组标识符[常量表达式]	
函数标识符( )	

上述初等表达式经各种运算符的操作得到表达式。表达式又可以和初等表达式一起再经运算符操作，得到的还是表达式。一般说来，表达式是初等表达式多次递归操作的结果。

### 3.3 表达式语句

1. 表达式语句是在表达式之后使用逗号终结符构成的语句。
2. 表达式语句有多种形式，共性是语句中只有表达式，例如：
  - 以赋值表达式为主构成的赋值表达式语句。
  - 由函数调用构成的函数调用表达式语句。在调用时，函数的形参表用实参表代换，而形

成表达式。

·由非左值表达式构成的表达式语句。

[例 3.1] 赋值表达式语句。

```
i = 5;
array[5][i] = 7;
leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
mystruct . salary = 200;
ptr = & array[0][0];
x = func(i, ptr);
```

超星阅读器提醒您：  
使用本复制品  
请尊重相关知识产权！

[例 3.2] 函数调用表达式语句。

```
func(i, ptr);
```

[例 3.3] 有函数原型为: `extern int printf ( char *format [ , argument , ... ] );`

下边是它们的调用例子:

```
printf ("The decimal value of x is %d . \n", x );
```

当  $x=100$  时: 执行后显示出: The decimal value of x is 100. 光标移下一行。

```
printf ("The hex value of x is 0x%x . \n", x );
```

当  $x=100$  时: 执行后显示出: The hex value of x is 0x64. 且光标移下一行。

```
printf ("The 0x 41 is ascii character%c . \n", x);
```

当  $x='A'$  时: 执行后显示出: The 0x 41 is ascii character A. 光标移下一行。

```
printf ("The content of buf is character string < % s > ", buf );
```

当 `buf [ ] = "Hello , welcome!"` 时: 执行后显示出:

The content of buf is character string <Hello , welcome! > 且光标移下一行。

### 3.4 复合语句

格式:

```
[说明表]
```

```
语句表
```

其中:

(1) 说明表是一个到多个用分号分割的说明的总称。说明表可有可无。

(2) 语句表是一个到多个分号分割的语句的总称。语句允许用复合语句代替, 形成复合语句嵌套。嵌套深度任意。

(3) 复合语句是由说明表和语句表用花括号对括起而构成。因为括在花括号对中, 结尾不必再加分号。

(4) 有的 C 语言, 允许在语句表中的任意位置插入在线汇编指令序列, 如:

```
[说明表]
```

```
语句表
```

```
[在线汇编指令序列]
```

语句表

[在线汇编指令序列]

...

|

(5) 复合语句中的语句又可以是另一个复合语句, 这种嵌套的复合语句原则上可以无限层次地进行下去。

## 3.5 循环语句

C 语言中循环语句共有三种形式:

- for 循环语句
- while 循环语句
- do while 循环语句

### 3.5.1 for 循环语句

格式:

for (表达式 1; 表达式 2; 表达式 3)

语句;

其中:

(1) 表达式解释如下:

- 表达式 1 是循环量赋初值语句;
- 表达式 2 是循环量终值的控制语句;
- 表达式 3 是循环量增值语句。

(2) 语句部分是 for 的循环体。语句也可以是花括号括起的复合语句。

(3) 执行 for 循环时, 首先用表达式 1 给循环量赋初值; 然后判表达式 2 是否满足循环条件, 满足则执行循环体中的语句部分; 语句部分执行完后, 再回来先执行表达式 3, 再表达式 2。如果满足循环条件则继续执行循环体的语句部分, 如此循环直到如不满足循环条件则结束整个循环语句。

[例 3.4]

```
void main( )
{
    int i;
    for ( i = 1; i < 4; i + + )
        printf ( " %d \n", i )
}
```

执行后显示器屏幕显示:

1  
2  
3

[例 3.5] 5 世纪时我国数学家提出“百鸡问题”: 鸡翁一, 值钱五; 鸡母一, 值钱三; 雏三, 值

钱一,百钱买百鸡,问鸡翁、母、雏各几何?

设公鸡、母鸡、小鸡各  $x, y, z$  个,有方程:

$$5x + 3y + z/3 = 100 \quad (3.5.1)$$

$$x + y + z = 100 \quad (3.5.2)$$

变量 3 个,方程式 2 个。变量大于方程数,为不定方程,有多个解。

用试算法先给定  $x$  和  $y$ ,用公式(3.5.2)求  $z$ ,再根据(3.5.1)式判  $z$  是否合理,合理的为解,不合理的弃之。

```
void main ()
{
    int x, y, z;
    for (x=0; x<=20; x++)
    {
        for (y=0; y<34; y++)
        {
            z=100-x-y;
            if ((z%3==0) && ((5*x+3*y+z/3)==100))
                printf("cock=%d\t hen=%d\t chicken=%d\n", x, y, z);
        }
    }
}
```

执行后显示器上显示:

```
cock = 0    hen = 25    chicken = 75
cock = 4    hen = 18    chicken = 78
cock = 8    hen = 11    chicken = 81
cock = 12   hen = 4     chicken = 84
```

本例里层 for 为复合语句,外层 for 也是复合语句。外层 for 循环量是公鸡数,初值应为 0, 100 元最多买 20 只,所以循环条件是  $\leq 20$ 。里层 for 循环变量是母鸡,初值为 0, 100 元买不到 34 只,所以循环条件  $< 34$ 。if 语句的条件表达式为二个关系表达式相与,一个是小鸡数应为 3 的倍数,另一个是买各种鸡钱的和数应为 100。考虑到  $=$  的优先级高于  $\&\&$ ,所以两个  $=$  关系表达式外边的圆括号都是可以省掉的,为保险起见,都加上了也不为错。

### 3.5.2 while 循环语句

格式:

```
while (条件表达式)
    语句;
```

其中:

(1) 条件表达式是循环是否能继续下去的条件。

(2) 语句是循环体。语句可以是单语句或复合语句。循环体每执行完毕一轮,就要重新检查一次条件表达式是假还是真,是真,继续循环下去,是假则结束本循环语句。

[例 3.6]

```
void main ()
{
    int i;
    i=1;
```

```
while (i < 4)
    printf ("%d\n", i++ * 10);
```

执行结果, 显示:

```
10
20
30
```

[例 3.7]

```
void main ( )
{
    int count;
    count = 0;
    while (count < 4)
    {
        count ++;
        printf ("%d\n", count);
    }
}
```

执行结束, 显示:

```
1
2
3
```

[例 3.8] 延时函数

```
void delay (int t)
{
    while (t)
        t --;
}
```

函数的参数为 t, 在函数内部循环做 t 减 1 运算, t 减到 0 结束循环返回。本例也可改为:

```
void delay (int t)
{
    while (t--);
}
```

### 3.5.3 do while 循环语句

格式:

```
do
    语句;
while (条件表达式);
```

其中:

(1) 语句可为复合语句。

(2) 控制循环结束的条件表达式放在循环体的后面。所以, do while 语句必定至少先要执行一次循环体。

(3) 每次执行完循环体之后, 重新计算一次条件表达式。条件为真, 转回去再执行一轮循

超星阅读器提醒您:  
使用本复制品  
请尊重相关知识产权!



环体,直到表达式非真才结束 do while 循环语句。

(4) 对于不明确究竟应该循环多少次,但至少要执行一次循环体的场合,适合使用 do while 语句。

[例 3.9] 写一函数将整型数转换为 ASCII 字符串。

```
void itoa (int n, char s[ ])
{ int i, sign, c;
  if ((sign = n) < 0)
    n = -n; /* 如为负数,则化为正数 */
  i = 0;
  do
  { s[i++] = n%10 + '0'; /* 个位,十位,...,转换为 ASCII 符 */
    while ((n = n/10) > 0) /* 或写做 while (( n/=10) > 0) */
  } if (sign < 0)
    s[i++] = '-'; /* 放符号位 */
  s[i] = '\0'; /* 放串的结尾符 */
  for (i = 0, sign = strlen(s); i <= sign; i++, sign--)
  { c = s[i]; /* 除结尾符不动外全串反转 */
    s[i] = s[sign];
    s[sign] = c;
  }
}
```

本例中:

(1) do while 部分先求出按个位、十位、百位...升序排列的 ASCII 字符串,如果是负数再放负号。

(2) 然后用 for 语句把 ASCII 字符串次序倒过来。如果是负数,先放负号字符,再是从最高位 ASCII 字符开始降序排列直到个位的 ASCII 字符。

(3) 在 for( ) 圆括号内,初值表达式语句和增值表达式语句都用了逗号运算符。在 for( ) 中的变量 sign 是临时借用,存放中间变量的。strlen(char \* ptr) 是函数库中定义的返回字符串长度的函数。

## 3.6 条件语句

C 中的条件语句有:

- 一般条件语句
- 嵌套条件语句
- 多选一条件语句

### 3.6.1 一般条件语句

格式有二种:

if (条件表达式) 或 if (条件表达式)

```

    语句 1;
else
    语句 2;

```

其中:

(1) if 后面的条件表达式的结果为真时, 执行语句 1, 执行完后跳过语句 2, 结束整个条件语句; 条件表达式的结果为假时, 跳过语句 1, 执行 else 部分的语句 2, 之后结束整个条件语句。

(2) 条件语句允许没有 else 部分, 意思是条件为假时, 什么也不做, 结束整个语句。

(3) 语句 1 和语句 2 可以是花括号括起的复合语句。

(4) 在 C 中以 0 为假, 非 0 为真。所以, C 语言中测试的是表达式的数字值, 只要是非 0 值就是真。为此:

if (表达式 != 0) 简写为 if (表达式)

[例 3.10] 试将条件表达式语句

```
z = a > b ? a : b;
```

改写为条件语句。

```

if (a > b)
    z = a;
else
    z = b;

```

### 3.6.2 嵌套条件语句

格式:

```

if (表达式 1)
{
    if (表达式 2)
        语句 1;
    else
        语句 2;
}
else
{
    if (表达式 3)
        语句 3;
    else
        语句 4;
}

```

其中:

(1) 外层条件语句的 if 部分被表达式 2 为特征的条件语句所嵌套, 外层条件语句的 else 部分被表达式 3 为特征的条件语句所嵌套。每部分的嵌套语句都放在花括号之中, 这样写法能保证不出二义性的问题。

(2) 二义性的问题: 因为条件语句的 else 部分可以没有, 在嵌套不合适时会出现二义性问题。例如, 外层 if 部分嵌套时未用花括号对括起, 嵌套的里层又正好没有 else 部分(语句 2), 那

么,下面的 else 部分就产生了二义:不知道它应归属于里层的 if 还是外层的 if。编译程序在这种情况下隐含的处理原则是“就近原则”。由于 else 靠近里层的 if 便作为里层 if 的 else 部分,正好与原意相背。所以,条件语句在嵌套时,如果可能出现二义性问题时,嵌套部分一定要放在花括号之内。

### 3.6.3 多选一条件语句

多选一条件语句也叫做散转语句,是将各种可能的分支全部罗列出来,选择其一执行之。其格式如下:

if (表达式 1)	或改写做:	if (表达式 1)
语句 1;		语句 1;
else if (表达式 2)		elseif (表达式 2)
语句 2;		语句 2;
else if (表达式 3)		elseif (表达式 3)
语句 3;		语句 3;
else		else
语句 4;		语句 4;

·这是一种每次都嵌套在 else 部分的多层嵌套条件语句。由于没有嵌套到 if 部分不会出现因嵌套语句缺 else 部分而产生二义。最后 else 的部分(即语句 4),根据“就近原则”归属于表达式 3 为特征的条件语句,正好是符合原意的。

·最后的 else 部分是所有分支都遍历后的多余部分,本是可有可无的。但是在 C 语言中,经常保留它,用来捕捉不该发生的出错情况。

·格式中左边的写法常被右边写法所代替。因为不会发生二义,右边的写法显得整齐而练达。

[例 3.11] 试编写求解二次方程根的实用程序。二次方程为:

$$ax^2 + bx + c = 0$$

写实用程序时要考虑方程式原始条件 a, b, c 的各种可能的情况,如:

- (1) 当  $a=b=0$  时,不论 c 是否为 0,方程将全面退化,从而无解存在。
- (2) 当  $a=0$  而  $b \neq 0$ ,退化为一次方程,只有一根。
- (3) 当  $a \neq 0$  才为二次方程。

求根的通式为:

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

讨论:判别式  $b^2 - 4ac > 0$  有二实根,若  $c=0$ ,有一为 0 根。

$b^2 - 4ac = 0$  有重实根。

$b^2 - 4ac < 0$  有二复根,若  $b=0$ ,变为二虚根。

源文件内容:

```
# include <stdio.h>
# include <math.h>
```

超星阅读器提醒：  
使用本复制品  
请尊重相关知识产权！

```
void    main ( )
{
    double a, b, c, discriminant, rel, im;
    scanf ( " %lf %lf %lf ", &a, &b, &c);
    if ((a == 0.0) && (b == 0.0))
        printf ( " \t the equation is degenerated. \n");
    elseif (a == 0.0)
        printf ( " \t single root only % .6f \n", -c/b );
    elseif (c == 0.0)
        printf ( " \t the roots are % .6f and % .6f \n", -b/a, 0.0 );
    else
    {
        rel = -b/(2.0 * a);
        discriminant = pow(b, 2.0) - 4.0 * a * c;
        im = sqrt (fabs(discriminant))/2.0 * a;
        if (discriminant >= 0.0)
            printf ( " \t the roots are % .6f and % .6f \n", rel + im, rel - im );
        else
            printf ( " \t the roots are complex % .6f + j % .6f " \
                    "and % .6f - j % .6f ", rel, im, rel, im );
    }
}
```

输入: 0.0 0.0 7.0

显示: the equation is degenerated.

输入: 0.0 10.0 2.0

显示: single root only -0.200000

输入: 2.0 3.0 0.0

显示: the roots are -1.500000 and 0.000000

输入: 1.0 5.0 6.0

显示: the roots are -2.000000 and 3.000000

输入: 1.0 1.0 1.0

显示: the roots are complex -5.000000 + j0.866025 and -5.000000 - j0.866025

讨论:

(1) 本程序开头用了 `#include` 预处理器伪指令。因为, 在程序中用到的函数 `printf()` 和 `scanf()` 的原型说明是放在头文件 `stdio.h` 中, 而 `pow()` 和 `sqrt()` 是放在头文件 `math.h` 中。因为要调用这些函数, 根据前面说过的规则, 必须加上以 `extern` 开头的原型说明, 现在包含了 `stdio.h` 和 `math.h`, 就等于加了原型说明。

(2) `scanf()` 是前面讲过的 `printf()` 的逆过程。后者是格式化输出, 前者是格式化输入。两者的参数格式相同, 用法也一样。 `scanf()` 要求通过标准输入设备(键盘)输入数字字符, 然后经它转换成计算机内部存放时的数值。以前说过, 作为参数部分的 `%f` 是指 32 位浮点数(float), 在 `f` 之前加 `l`, 即 `%lf` 为 64 位双精度浮点数(double)。参数部分的 `%.6lf`, 其小数点后的 6 说明在转换成数值时, 要求小数点后要保持 6 位数。整数部分未规定, 实际有几位算几位。

### 3.7 开关语句

开关语句是解决多路分支问题的有效方法。它的作用与前面介绍过的多选一 if 条件语句一样,只不过更为直观而灵活。

开关语句格式如下:

```
switch (整型数表达式)
{
    case 常量表达式 1: [语句 1;]
    case 常量表达式 2: [语句 2;]
    ...
    case 常量表达式 n: 语句 n;
    [default: 语句 n+1;]
}
```

其中:

(1) switch 的表达式计算结果一定是 int 整型数(包括字符值)。

(2) 表达式中可用的运算符包括:

单目: - ~

双目: + - \* / % & < < > > == != < <= > >=

三目: ? :

还有: sizeof ( )

(3) 用于 case 中常量表达式 1, ..., 常量表达式 n 的类型, 与 switch 表达式的类型相同。switch 表达式的可能结果, 应与 case 的常量表达式 1~n 有一一匹配的关系。

(4) 语句 1, ..., 语句 n+1, 可以是复合语句, 也可以是空(即没有语句)。

(5) 冒号前的 case 部分叫做情况语句前缀。

(6) 冒号前的 default 是缺省情况前缀。凡未包含于前面的一切其他情况, 都由 default 部分予以处理。如果罗列的 case 将包含所有的情况, default 情况部分可以不要。但是在 C 中, 常常将它保留, 用来捕捉异常情况, 并给出提示或应急处理。

(7) switch 语句执行时, 先计算表达式, 按其结果的 int 常量与 case 的常量表达式相匹配。匹配上的就转去该 case 执行; 匹配不上的, 则转去 default 情况执行。如果没有 default 情况部分, 就结束本 switch 语句。

(8) 冒号之前的情况前缀, 实际上是语句的标号。所以, 控制转到某一情况语句可往下顺序执行。换句话说, 它可以不理睬下面的情况标号, 贯通执行下面情况中的语句, 直到遇到控制转向语句。switch 语句中可用的控制转向语句有 break 语句和 goto 语句。

[例 3.12] 编写一个程序, 要求由键盘输入一段正文, 然后统计该正文中每个数字字符('0'~'9')的出现次数, 空白符(nul)出现次数和其他字符的出现次数。

本例中:

(1) 使用函数 getchar( ) 由标准输入设备取得字符值放入变量 c 中, c 作为 switch 语句的表达式, 与 case 的各字符常量相匹配。数字字符 '0'~'9' 处理的方法是一样的, 所以安排在一个 case 之中。但是, 在向数组 digit[10] 相关元素中累加次数的时候, 需要找准位置。程序

中 `digit[c - '0']` 的下标部分,是用数字字符值求下标值。

(2) `break` 语句是使控制转出 `switch` 语句(`break` 语句用法详情见后)。

(3) `switch` 的外层是 `while` 循环语句,等待再次输入字符。只要未输入 `ESC` 键,就一直循环。输入 `ESC` 键,则结束 `while` 循环并打印统计结果。之后,结束程序。

(4) `switch` 语句中的空格符、回车符和制表符都归为空白符类进行统计处理;输入的其他字符,归入 `default` 类进行统计。

源文件内容:

```
# include <stdio.h>
# define EOF '\27'
void main()
{ int c, digit[10], i, white, others;
  white = others = 0;
  for (i = 0; i < 10; i++)
    digit[i] = 0;
  while ((c = getchar()) != EOF)
  { switch (c)
    { case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9': digit[c - '0']++;
                break;
      case ' ':
      case '\n':
      case '\t': white++;
                break;
      default: others++;
                break;
    }
  }
  printf("statistics of digits from 0~9:");
  for (i = 0; i < 10; i++)
    printf("%d, ", digit[i]);
  printf("\n white space = %d, others = %d\n", white, others);
}
```

程序中用了 `getchar()` 函数调用,需要用 `stdio.h` 中的相关原型说明。另一个预处理器伪



指令 #define 将 ESC 键的字符值(\27)用标识符 EOF 代替,说明是一轮正文输入的结束。

### 3.8 间断语句

格式:

```
break;
```

break 专门用于循环语句(包括 for 循环、while 循环和 do while 循环语句)和 switch 语句之中。执行 break 语句,控制直接跳出所在层的循环语句或 switch 语句。

[例 3.13] 已知工厂产值的年增产率,编写一个程序求产值翻番需要多少年?工厂产值的年增产率由键盘输入。

设当前产值为 100.0,源文件的程序部分如下:

```
void main( )
{ float productivity, p0 = 100.0, rate;
  int year;
  for (;;)
  { printf("please input increase rate:")
    scanf(" %f", & rate)
    if ( rate <= 0.0 || rate >= 1.0 )
      break; productivity = p0;
    year = 0;
    for (;;)
    { year + + ; productivity * = (1 + rate);
      if (productivity > 2 * p0)
      { printf("rate = %f \t double productivity duration " \ "%d years. \n", rate, year);
        break
      }
    }
  }
}
```

讨论:

(1) for 循环语句的 for 循环头是空的,即循环变量的初值、结束条件及循环量增值语句均没有。这说明,它是个无穷循环语句。

(2) 外层的无穷循环是不断地等着用户由标准输入设备输入年增长率,每输入一次,程序作一次增长率检查,增长率应在 0.0~1.0 之间,不包括 0.0 及 1.0。凡是不在此范围的,就用 break 语句跳出外层 for 无穷循环,从而结束程序的执行。

(3) 如果增长率在合理范围内则为里层 for 无穷循环赋初值,并进入里层的无穷循环。每循环一次,年数加 1,并累进计算一次当年的产值。当产量达到最初产值一倍的时候,在执行输出语句之后,用 break 语句跳出内层无穷循环,回到外层无穷循环,要求用户再次输入年增长率,再次求翻番产值所需年数。

(4) 如果要结束程序,只要给不合理的增长率即可,如 0.0 增长率。



### 3.9 接续语句

格式:

```
continue;
```

作用:

(1) continue 语句只能用于循环语句的循环体内(包括 for 循环、while 循环和 do while 循环)。

(2) 执行 continue 语句的作用是:控制立即跳过其后全部存在的语句而直接进入下一轮新循环的开始。

[例 3.14] 有如下 for 循环,在循环的复合语句中,当数组元素为负值时不做处理,跳过下面各语句,直接进入下一轮新的循环。

```
for(i=0;i<n;i++)
{if ( a [i]<0)
    continue;
    ...
}
```

上面这段程序也可改为数组元素为非负值才做处理。

```
for(i=0;i<n;i++)
{if (a [i]>=0)
    ...
}
```

两段程序作用一样。后者省去使用 continue 语句,但多一层嵌套关系。前者正好相反。

### 3.10 跳转语句

格式:

```
goto 标号;
```

其中:

(1) 标号是标识语句地址的标识符。

(2) 跳转语句执行时,将控制转到标号所在的语句,并继续执行之。

(3) C 语言允许复合语句的无限嵌套。各嵌套层又允许起重名的标号。因此,跳转语句的转入目标会产生多义。和其他语言一样,C 语言规定:

- 跳转不能向内层跳。

- 在本层或外层上有同名标号时,规定只跳到最近一层的标号处。

- 当然,同一层是绝对不允许重名,必须保持各标号的惟一性。

(4) goto 语句可以向外穿透多层转移控制。这是结构型语言不允许存在的现象。所以,从 1968 年开始,对于 goto 语句曾掀起热烈而深入的讨论,直到 1974 年才作出了结论。允许

这个反结构的 goto 语句,作为特例,存在于结构语言之中。理由正是因为它具有结构语句所不能直接得到的多层穿透性。请注意,结构性语言不是不能做到多层次地转移,而是不能直接了当的穿透。下面看一个例子。

[例 3.15] 寻找某一二维数组中的第一负值元素,找到后直接将控制转出二重循环。

```
for (i=0;i<N;i++)
    for (j=0;j<M;j++)
        if (v[i][j]<0)
            goto found;
...          /* 放未找到负值元素时应执行的语句 */
found:      /* 找到负值元素应执行的语句 */
...
```

例中用 goto 语句向外转移二层 for 循环,到达标号 found。

[例 3.16] 不用 goto 语句,改用结构语言完成上例同样的操作。

```
int negative;
negative=0;
for (i=0;i<N && ! negative;i++)
    for (j=0;j<M && ! negative;j++)
        negative = v[i][j]<0;if (negative)
...          /* 找到负值元素时应执行的语句 */
else
...          /* 未找到负值元素时应执行的语句 */
```

由本例可以看到如果用结构语言完成上例同样的操作,则:

- 一要引进新变量 negative 作为标志。
- 二要一个元素一个元素地为 negative 置入 0 与非 0 的标志。
- 为要能在第一次 negative 置入非 0 的标志,就能及时双层结束循环,在双层循环的每一层都要额外设置检查标志 negative 的条件,只要 negative 置非 0,即终止循环。
- 在找到负值元素后的下一步,再使用 if 语句分别执行找到负值元素与未找到时的语句。
- 由上可见,不用 goto 语句,便要引入新变量并要不断循环地执行对变量的置标志,程序开销很大而且可读性与可维护性都不好。
- goto 语句不受结构语言的约束也的确容易发生意外,所以应尽量少用和慎用。

## 3.11 返回语句

C 语言中返回语句有两种格式:

```
return;
return(表达式);
```

其中:

- (1) 前者执行时,将控制返回调用程序。
- (2) 后者执行时,在返回控制时还将表达式的结果按照函数要求类型回代给调用程序。
- (3) 返回语句只能用于函数之中,且在函数中可以多次使用。

[例 3.17] 编写一个函数把数字字符串转换为双精度浮点数。

```
double atof(char s[])
{
    double val, power;
    int i, sign;
    for (i=0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t'; i++) ;
        /* 跳过所有的前导空白符 */
    sign = 1;
    if (s[i] == '+' || s[i] == '-')
        sign = (s[i++] == '+') ? 1 : -1;
    for (val = 0; s[i] >= '0' && s[i] <= '9'; i++)
        val = 10 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1; s[i] >= '0' && s[i] <= '9'; i++)
    {
        val = 10 * val + (s[i] - '0');
        power *= 10;
    }
    return (sign * val / power);
}
```

讨论:

(1) 把字符数组  $s[i]$  中的 ASCII 数字字符串转换成双精度浮点数并返回之。在 C 语言中, 浮点数在计算机内部一律转换为双精度数。所以, 函数返回类型最好用 double 不要用 float。这样, 还要直接一些, 精度也高些。

(2) 程序中, 先用 for 的空循环语句把 ASCII 串中可能存在的前导空白符均略去, 直到出现第一个非空白符。

(3) 接下去, 先判符号位并放在变量 sign 中。再用 for 循环语句不断地取数字字符 '0' ~ '9' 并转换为数字 0~9, 加到上次 val 值的 10 倍之上, 构成新的 val 值, 一直进行到下个非数字字符。

(4) 再下去, 应检查小数点。有小数点, 继续向下取小数部分, 直到非数字字符。每次取得小数点后的数字字符暂时仍按整数部分一样处理, 求得的 val 值比真正的值每次都多乘了 10。为此, 在处理每个小数字符的同时, 把初值为 1 的 power 变量自乘一次 10。如此, 将最后所得 val 除以 power 即应是真正值的 val 值。

```
或 int strlen(char *p)
{ int n;
  for (n = 0; *p != '\0'; p++)
    n++;
  return(n);
}
```



两种方法完成的操作虽然相同,但是做法上有区别。

(1) 上例的前一种做法是:

·把字符数组的首地址作为参数传进函数。函数内部定义了一个指针。用这个传进来的首地址为指针赋初值。

·随后,在 while 语句中不断修改这个内部的指针,循环的条件取为指针指向的内容不是串的结尾符。所以,while 语句的指针值减去指针初值,便是字符串的长度。把它放在 return() 语句的括号中返回。

(2) 上例的后一种做法是:

传给函数的指针应事先已经指到了数组头部。因 C 语言是向函数传值的语言,所以,传给函数的这个指针只不过是内存中这个指针的复制品。所以,在函数中可以实行改变其值的操作。在 for 语句中不断用这个指针寻找字符串的串尾符。同时,令变量 n 不断加 1,直到字符串结束。这时的 n 值便是字符串的长度,放在 return() 语句中返回。

后一方法也可按前法进行修改如下:

```
int strlen(char *p)
{ char *p0;
  p0 = p;
  while (*p0 != '\0')
    p++;
  return (p - p0);
}
```

还有一种看起来更简捷的写法:

```
int strlen(char s[])
{ int i = 0;
  while (s[i++]);
  return (i);
}
```

### (3) 空语句的价值:

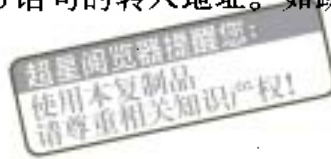
在不应该放实质性语句的地方加上标号,作为 goto 语句的转入地址。如跳转到复合语句的最后。

```
| ...  
    goto end;  
    ...  
end;  
|
```

给空函数占一个入口地址,函数本身则留待以后补写,如:

```
void func1()  
{  
  
};  
|
```

对于中断服务程序,需要把它的入口地址填入中断向量表。这样就可以先写一个空的中断函数,以取得地址。提前向中断向量表进行填表。





## 第四章 函数及函数库

使用本文档  
请尊重相关知识产权!

### 4.1 前言

C 语言的程序是由一个主函数和一系列函数组成的。主函数和函数又是由说明和语句组成。说明是数据部分,语句是程序部分。说明和语句全部集中在函数(包括主函数)之中。所以,C 语言是关于函数的语言。其他高级语言中的过程与函数的概念,在 C 语言中全部归并为函数。实际上,C 语言中无返回量的函数就是其他高级语言中的过程。

C 语言程序设计提倡把大任务分解为小的、充分独立的、功能单一的函数,以提高程序的可读性、重用性、易维护性和可移植性。函数把操作的细节隐蔽在函数内部,尽可能地排除细节问题对程序员主体思路的干扰。

C 语言还提供了许多常用的关于输入/输出、类型转换、数值计算等函数组成的运行时间库,供用户使用,使用户不必事事从头做起,从而大幅度地减少开发的时间。随着 C 语言一并提供的函数库无须再行编译,它是以目标码形式提供的,所以叫做运行时间库。它是在连接时自动从函数库中找出并链接到程序之内的。运行时间库库函数的原型说明照例是放在相应的头文件中。为了引用库中函数,无须用户罗列所需引用库函数的原型说明,而是在程序中用预处理器伪指令将有关的头文件包含进来即可。

关于用户自己编写的函数,应如何做函数的定义性说明和原型说明,前面已有论述。本章主要介绍函数的调用以及函数调用的实例。C 语言是一个表达式非常丰富的语言,其语句种类虽不多(只有 10 种),但允许多层次的嵌套,使得语句编写的技术性很强。C 语言灵活高效、功能极强、简单优美、生动活泼,因此发展很快。为了尽快地提升读者的水平,本章举例不是以照顾一般水平为出发点的。初学者遇到未能立刻理解的问题时,尽可暂时放弃,待到有过一些经验时再来试读,可能要多次反复,这并不为怪。但是,每重复一次,相信定会有新的收获。

### 4.2 单文件程序(一)——字符串处理

作为例子编写一个单文件程序,解决下述问题:由键盘不断送入一行一行的正文,每行以换行符作为结束。每敲完一行,便与以前输入过的最长行做比较,并把截止到当前最长的行显示出来。

1. 把上述要求组织成多个函数,并用一个主函数统帅起来。全部函数,包括主函数放在一个文件中形成单文件程序。

2. 可写这样几个函数:

- 应有能识别行的结束,并把输入行的内容暂时保存下来的函数。起名叫 `getline`,它应返回本行的实际长度。

- 应有一个函数,能把迄今为止最长行的内容复制到另一个缓冲器中,以便以后显示之用。

起名 copyline。

- 要有从键盘读入字符的函数,使用库中函数 getchar。
- 要有显示函数,不必自编,使用库中函数 printf。
- 最后要有主程序 main,按题意把上述函数组织起来。

源文件如下:

```
# include <stdio.h>
# define MAXLINE 80          /* 输入行最长控制在 80 个字符 */

extern int  getline (char curline [ ], int limit );
extern void copyline (char maxline [ ], char curline [ ]);

void main()
{
    int limit = MAXLINE;
    int len;
    int max;
    char curline [MAXLINE];
    char maxline [MAXLINE];
    max = 0;
    while ( (len = getline (curline [ ], MAXLINE )) > 0)
    {
        if (len > max)
        {
            max = len;
            copyline ( maxline [ ], curline [ ]);
        }

        printf ("so far the longest line is %s. ", maxline);
    }

    int  getline (char curline [ ], int limit )
    {
        int i, c;
        for ( i=0; i<limit-1 && (c = getchar ( )) != '\n' && c != '\27'; i++)
            curline [i] = c;
        if (c == '\n')
        {
            curline [i++] = c;
            curline [i] = '\0';
            return (i);
        }

        void copyline (char maxline [ ], char curline [ ])
        {
            int i;
            i = 0;
            while ((maxline [i] = curline [i]) != '\0')
```

```
i++;
```

### 4.3 单文件程序(二)——二维数组

编写一个程序：已知某年某月某日，求该天在当年中累计的正确天数。写这个程序的要点是必须正确处理大小月问题和闰年对二月天数的影响问题。因为问题比较简单，写成单文件程序。内容如下：

- 写一个函数，输入年、月、日，返回年中的累计日数，起名为 `yearday ( )`。

- 主函数应提示用户分别输入年、月、日，然后调用求年内累计日数的函数 `yearday ( )`，并将结果显示在屏幕上。

- 将正常年中每月的天数按大小月排成一维数组，再把闰年每月的天数按大小月排成另一个一维数组。把上述的两个一维数组连排起来组成一个二维数组。根据闰年和正常年就可以容易地由这个二维数组得到每月的正确天数。最好是把这个二维数组作为外部变量处理，以方便其他函数使用。

单文件程序如下：

```
# include <stdio.h>
extern yearday (int year, int month, int day) ;
int mothday [2][13] = {{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
                       {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};

void main ( )
{int y, m, d;
    printf ( "please give year:" );
    scanf ( "%d \n", &y);
    printf ( "please give month:" );
    scanf ( "%d \n", &m");
    printf ( "please give day:" );
```

(2) 在 leap 开头的表达式语句中, 赋值运算符右侧的结果是逻辑量 0 或非 0。它们自动向左值的 int 类型转换后为 0 或 1。请注意  $=$ ,  $!=$  的优先级高于  $\&\&$ ,  $\&\&$  又高于  $\parallel$ 。

(3) 二维数组 monthday[2][13] 对函数 yearday() 是外部定义的全局变量, 在函数中可见。它在本文件中定义而且定义在函数之前, 所以, 函数对它不需加引用性的说明, 就可直接存取。

## 4.4 多文件程序——台式计算器逆波兰算法的实现

台式计算器是用台式计算机实现的计算器功能。本例实现的是关于算术四则运算的计算器。运算符有  $+$ ,  $-$ ,  $*$ ,  $/$  和  $=$ 。计算器能够接受最多 20 位十进制数字, 小数点包括在内。任何时候按 “c” 键可以清零, 以等待新一轮求算。计算的实现采用逆波兰法, 即把下面的求算:

$$(2-3) * (5+6) = \quad (4.4.1)$$

转变为:

$$23-56+* = \quad (4.4.2)$$

这种转变的具体实现如下:

- 使用两个堆栈。按 (4.4.1) 式, 遇到数字字符 (包括小数点) 转换成为浮点数压入第一个栈——数值栈内。遇到非数字的运算符等暂时压入第二个栈——算符栈内。

- 每次由标准输入设备取到的若是数字字符, 暂时存入缓冲器中, 待完整的数据取得后, 通知主函数。在那里, 先转换成数值随即压入数值栈。

- 每一轮取字符时, 先去算符栈取算符, 如果算符栈空, 则从标准输入设备输入字符。取到的字符如果是数字同上处理。如果是算符字符, 即  $+$ ,  $-$ ,  $*$ ,  $\div$  便由数字栈弹出两个数字进行相应的运算, 相当于变成 (4.4.2) 式的运算。最后, 遇到  $=$ , 便显示结果值。如果是  $($ ,  $)$  字符, 则忽略。如果是字符  $c$  则清屏幕, 等待下一轮求解。

本程序分四个源文件。它们是:

- 文件 1——放一个关键函数, 专做分拣工作。它由标准输入设备或算符栈取得数字字符或非数字字符。如果是数字字符, 取完数字字符串后置标志, 等待主程序处理。数字字符串超过 20 位十进制数, 置溢出标志, 等待主程序处理。非数字字符则压入算符栈。

- 本程序需要有一个缓冲器存放分拣出来的数字字符串, 供主函数转化为浮点数压入数字栈。这个缓冲器应由本程序和主程序共享。为了安全, 不采用全局变量的方案, 而是仅定义于主程序的内部, 但将它的首地址作为参数传送给函数。

- 文件 2 和文件 3——分别放数字栈和算符栈的管理函数, 把各自的栈放在它们自己的管理函数之前。把各个栈和管理它的函数对外独立起来, 可以减少被意外的修改, 做到数据安全和实现程序坚固的目的。计算器是涉及财务进进出出的工具, 数据的准确和不受干扰是同等重要的。用 C 语言做开发, 不怕源文件多, 文件越多, 开发的效率越高 (因为有 MAKE 实用程序的支持)。

- 文件 4——放主程序。主程序虽然是程序中最先进入并执行的部分, 但是, 不在乎放在哪个文件中或者在文件的前部还是后部。主函数虽然是程序的首脑, 但是, 在本例中它仅能访问定义在它内部的临时存放数字字符串的缓冲器。其他二个堆栈对它都是保密的, 它不可以

直接去访问。如同作为银行的行长,也并不能自己直接去访问金库一样。C 语言经历了软件危机的沉重教训,发展了自己的一套严密的组织管理手段。但是,要做到领悟,有一个修炼的过程。下面写出具体程序的大框架(并非最后的实用程序)。

#### 1. file 1

```
# include <stdio.h>
# define EOF      '\27'          /* ESC */
# define DATA    0
# define TOOBIG   9
extern char getchar ();
extern void storeop (char c);

int splitinput (char buff [ ], int limit) /* 本函数取字符输入,并分拣数字串放 buff [ ] */
{ int i, ch;
  if ((ch = getchar ()) == EOF) /* 操作符等压入操作堆栈 opstack [ ] */
    return (EOF);              /* 本函数专供主程序调用 */
  while ((ch = getchar ()) == ' ' || ch == '\t' || ch == '\n');
                                /* 去掉空白符 */
  if (ch != '.' && (ch < '0' || ch > '9'))
    return (ch);
  buff [0] = ch;
  for (i = 1; (ch = getchar ()) >= '0' && ch <= '9'; i++)
    if (i < limit)
      buff [i] = ch;
  if (ch == '.') /* 读小数部分 */
  { if (i < limit)
    buff [i] = ch;
    for (i++; (ch = getchar ()) >= '0' && ch <= '9'; i++)
      if (i < limit)
        buff [i] = ch;
  }
  if (i < limit)
  { storeop (ch);                /* 非数字字符压入操作堆栈,其中包括加减乘除等 */
    buff [i] = '\0';            /* 结束数字串 */
    return (DATA);              /* 返回数字串标志 */
  }
  else /* 如超过 buff 极限应做如下处理 */
  { while (ch != '\n' && ch != EOF)
    ch = getchar ();            /* 超过限定的数字全部? 到不送为止 */
    buff [limit - 1] = '\0';    /* 用终结符终止 ASCII 串 */
    return (TOOBIG);           /* 返回送数太多 */
  }
}
```



## 2. file 2

```
# include <stdio.h>
# define MAXSTACK128      /* 栈深度 */
                          /* 局部于文件的外部变量 */
int sp = 0;               /* 栈指针 */
double datastack [MAXSTACK];
```

```
double push (doublef )
{ if (sp < MAXSTACK )
    return ( datastack [ + + sp ] = f )
else
    { printf ( " stack full error. \n" );
      clear ( ) ;
      return (0);
    }
}
```

```
double pop ( )
{ if (sp > 0)
    return ( datastack [ sp - - ] );
else
    { printf ( "stack emptyerror ( \n" );
      clear ( ) ;
      return ( 0 );
    }
}
```

```
void clear ( )
{
    sp = 0;
    clrscr ( ) ;
}
```

## 3. file 3

```
# include <stdio.h>
# define MAXSTAC64
      /* 局部于文件的外部变量 */
char    opstack [ MAXSTAC ];
int      sptr = 0;

void     storeop (charc)
{
    if ( sptr = MAXSTAC )
        printf ( "operator stack full error. \n" );
    else
        opstack[ + + sptr ] = c;
```





```
char getcharacter ()
```

```
return ( spt > 0 ? opstac [spt - -] : getchar ( ) );
```

#### 4. file 4

```
# include <stdio.h>
```

```
# define MAXBUF 16
```

```
# define DATA '0'
```

```
# define TOOBIG '9'
```

```
# define EOF '\27'
```

```
extern double push (double f);
```

```
extern double pop ( );
```

```
extern void clear ( );
```

```
extern double atof (char buf [ ] );
```

```
extern int splitinput (char buff [ ], int limit);
```

```
void main ( )
```

```
int type;
```

```
char buf [MAXBUF];
```

```
double op2;
```

```
while ( ( type = splitinput (buf [ ], MAXBUF) ) != EOF )
```

```
{ switch (type)
```

```
    | case DATA: push (atof (buf)); /* 数据转换为浮点数压入 datastack */
      break;
```

```
    case '+': push ( pop ( ) + pop ( ) );
      break;
```

```
    case '*': push ( pop ( ) * pop ( ) );
      break;
```

```
    case '-': op2 = pop ( );
      push ( pop ( ) - op2 );
      break;
```

```
    case '\': op2 = pop ( );
      if (op2 != 0.0)
        push ( pop ( ) / op2 );
      else
```

```
        printf ( "zero divisor error. \n");
      break;
```

```
    case '=': printf ( "\t%f\n", push (pop ( ) );
      break;
```

```
    case 'c': clear ( );
```



```
case '(':
case ')': break;
case TOOBIG: printf("date too long (\n");
             break;
default:     printf("unknown command%c\n", type);
             break;
```

## 4.5 关于函数参数值的传递问题

C语言中,用参数形式向函数内部传送的对象标识符并不是内存中对象的本身,而是复制的对象付本。这叫做值传送。这个付本在函数内部是可以作为变量任意进行加1、减1或施行其他改变其值的操作。正因为改变的是对象的付本,所以内存中的对象本身并未随函数内部对其付本的修改而变化。这是值传送的很大优点,它保证了内存中数据的安全性。但是,值传送也有它未随人意的副作用。请看例4.1,例中是想将两个内存对象的值交换一下,一个是变量 $x$ ,一个是变量 $y$ ,进行交换的函数叫`swap()`:

[例4.1] `void swap (int x, int y)`

```
{ int temp;
  temp = x;
  x = y;
  y = temp;
```

在函数内部 $x$ ,  $y$ 确实是交换了,但是内存中实在的变量 $x$ ,  $y$ 的内容却丝毫未变。原因是传给函数是变量 $x$ ,  $y$ 的副本。然而,我们真正想交换的是内存中 $x$ ,  $y$ 的本身,这又当如何实现呢?办法是利用指针。把 $x$ ,  $y$ 的地址作为指针传给`swap()`。即

```
void swap (int *px, int *py)
{ int temp;
  temp = *px;
  *px = *py;
  *py = temp;
```

调用时,用`swap(& $x$ , & $y$ )`;传给函数的是 $x$ ,  $y$ 的地址,在函数的内部用指针加地址符“\*”实现对地址内容的交换。可见,如果想在函数中改变内存对象的值,需要把这个对象的地址代替指针参数送入函数。这是方法之一。其他方法,自然还有把该对象说明为函数外部的对象,令它在函数内部是可见的。只要函数定义性说明在这个外部对象定义的后面,就可以在函数内部直接存取;否则,需先加引用性的说明。至于引用性的说明放在函数之内或外都是可以的。

## 4.6 主函数

C 语言中,每个程序必须有且只能有一个主函数。它是程序执行的入口点。主函数格式如下:

类型说明符	main ( 参数表 ) { 语句 }
$\left\{ \begin{array}{l} \text{int} \\ \text{void} \end{array} \right\}$	$\left\{ \begin{array}{l} () \\ ( \text{void} ) \\ ( \text{int argc, char * argv [ ] ) \\ ( \text{int argc, char * argv [ ], char * env [ ] ) \end{array} \right\}$

其中:

- (1) 类型说明符指的是返回量的类型。对于主函数来说,只能是 int 和 void 之一。
- (2) 主函数的标识符只用小写的 main。
- (3) 参数表只能使用上边列出的四种之一。

·第一种和第二种是一样的,表示没有参数。

·第三种参数表的第一个参数,是命令行上给出命令字符串和参数字符串的总串数;第二个参数,是字符指针数组,由上述字符串首地址所组成。

·第四种参数表,比第三种多了一个指针数组的参数。这个指针数组是由程序赖以运行的系统环境字符串的首地址组成的。这个指针数组的长度是可变的,具体多少由系统环境块当时所放的环境变量的实际数来决定。实用中用得较多的是前三种参数表。

[例 4.2] 写一个程序,将执行该程序时键入的命令字符串、参数字符串以及当时的系统环境变量全部显示出来。程序放在名为 showarg.c 文件之中。

```
/* showarg.c */
#include <stdio.h>
#include <stdlib.h>

int main ( intargc , char * argv [ ] , char * env [ ] )
{ int i;

  printf ( "\t\tThe value of argc is %d . \n" , argc );
  printf ( "\t\tThese are %d command line arguments" \
    "passed to main: \n" , argc );
  for ( i=0; i<argc; i++ )
    printf ( "\t\targv [ %d ] : %s \n" , i , argv[i] );
  printf ( "\t\tThe environment string (s) on this system are: \n" );
  i=0;
  while ( env[i] != NULL )
    printf ( "\t\tenv [ %d ] : %s \n" , i , env[i++ ] );
  return (0);
}
```

一般 C 语言的源程序经编译并连接后,绝对目标文件按源文件主名命名。而在执行程序

时,命令行上只输入绝对目标文件名即可。以本例为例,main 所在文件名为 showarg . c;但在命令行的催促符后,敲入 showarg 即可。命令行上的参数是在程序名后以空白符分隔的一些字符串(包括单字符的串)。看下面的例:

在 DOS 的命令行催促符 c: \ DOS> 之后键入程序名和参数如下:

```
c: \ DOS> showargfirst - arg "this arg include blanks" 1 2 34 stop!
```

注意:命令和参数字符串中不允许有空格出现,因为空格已被用做参数之间的分隔符使用。但是,用双引号括起的字符串之中允许有空格。如本例中的第二个参数。上述的命令行,在 showarg 程序被执行后,显示出下列内容:

```
The value of argc is 7.
```

```
There are 7 command line arguments passed to main:
```

```
argv [0]:c: \ dos \ showarg
```

```
argv [1]:first - arg
```

```
argv [2]: "this arg include blanks"
```

```
argv [3]:1
```

```
argv [4]:2
```

```
argv [5]:34
```

```
argv [6]:stop!
```

```
The environment string (s) on this system are:
```

```
env [0]:COMSPEC = c: \ command.com
```

```
env [1]:PROMPT = $ p $ g
```

```
env [2]:PATH = c: \ dos; c: \ dos \ borlandc; c; d: \ windows
```

讨论:

(1) 主函数的返回值类型为 void 时为无返回值。这时主函数的函数体中不应有 return 语句。

(2) 返回值类型为 int 时,返回整数型状态码,主函数的函数体中应有带参数的 return 语句。

(3) 一般状态码是 0 为正常返回,非 0 为有错返回。错误原因按状态码值予以区分。返回的状态码,一般来说,供 DOS 的出口函数处理。用户也可以把非 0 的出错状态码交给自己编写的退出函数 voidexit(int status)处理。处理后再由 return(0)正常退出。自己编写的退出函数须经 atexit()函数登记后方可使用。

[例 4.3]

```
int    main ( )
{
    ...
    if ( )
        ...
    else
    { printf ("Error!");
      return (2);
    }
    return (0);
}
```

```
或  
  
int    main ( )  
{  
    ...  
    if ( )  
    ...  
    else  
    { printf ("Error!");  
      exit (2)  
    }  
    return (0)  
}
```

登记函数 `atexit()` 的使用未给出, 欲知详情请见有关库函数手册。

## 4.7 C 语言的函数库

C 语言一般都提供大量的关于 I/O 操作、内存分配、字符串操作、文件管理、数据类型转换、数学计算等函数库。库中除了符合 ANSI C 标准的库函数外, 一般都还有 C 编译器自己的扩展库函数部分。扩展的库函数部分, 各 C 编译器的生产厂家是不尽相同的。为此, 库函数的介绍只能以具体的 C 编译器为例。下面仅以 Borland C++ 支持 DOS 的函数库为例做介绍。全部库函数放在 `Cx.LIB`, `MATHx.LIB` 和 `GRAPHICS.LIB` 文件中, 文件名中的 `x` 为 `t`, `s`, `c`, `m`, `l`。它们分别对应于 `tiny`, `small`, `compact`, `midium`, `large` 和 `huge6` 种存储模式。C 语言提供的函数库是以执行代码的形式出现的, 供用户连接定位时使用。一般情况下, 生产厂家若声明所提供的函数库属开放系统的话, 则还应提供库函数源文件。运行时间库的函数原型说明是分散在不同的头文件中的。第 4.8 节列出 Borland C++ 的各种头文件。第 4.9 节分类列出库函数的函数名及隶属的头文件名。它们的列出仅为读者迅速浏览之用, 并非需要立刻掌握。读者可以利用它们, 隔一段时间浏览一遍, 作为测试自己当前水平的手段。至于具体函数的内容和示例因超出本书的范围, 请阅读《BORLAND C++ 库参考指南》。

## 4.8 头文件

头文件也有叫做包含文件。头文件包括库函数用到的: 符号常量的定义、复合数据类型和自定义数据类型的原型、库函数的原型说明和全局变量的说明等。它们是引用库函数必不可少的。头文件是厂家必定提供的, 用户可以只有在需要时查阅, 若非必要, 尽可不看。关于库函数的说明和示例则是需要下些功夫细读的, 但是必需是用什么, 看什么, 坚持用中提高的原则。库函数的说明和示例太多, 请详见有关的库函数参考手册。这里因为篇幅问题未能列出。表 4.1 是按字母排列的 ANSI C、Borland C 和 Borland C++ 头文件的内容简介。

表 4.1 ANSI C, Borland 和 Borland C++ 头文件中内容

ANSI C	assert.h	assert 调试宏
ANSI C	ctype.h	字符分类和转换
ANSI C	error.h	错误码定义宏
ANSI C	float.h	浮点数参数宏
ANSI C	limit.h	整型数范围, 环境参数和时间限制
ANSI C	local.h	国家语言等信息
ANSI C	math.h	数学函数原型
ANSI C	setjmp.h	定义结构类型 jmp-buf
ANSI C	signal.h	signal, raise 函数原型
ANSI C	stdarg.h	读变参数表宏
ANSI C	stddef.h	定义公共数据类型和宏
ANSI C	stdio.h	标准 I/O 预定义流和流级函数原型
ANSI C	stdlib.h	标准常用函数原型(转换、搜索、排序等)
ANSI C	string.h	串和内存函数原型
ANSI C	time.h	定义时间结构类型和时间函数原型
C	bios.h	IBM PC rom bios 函数原型
C	conio.h	DOS 控制台操作函数原型
C	alloc.h	内存动态管理函数原型
C	dir.h	目录、路径的宏、结构类型和函数原型
C	direct.h	目录、路径的宏、结构类型和函数原型
C	dirent.h	POSIXD 目录、路径的宏、结构类型和函数原型
C	dos.h	DOS 和 8086 的宏、结构类型和函数原型
C	fentl.h	与库函数 open 有关的定义宏
C	graphics.h	图形函数原型
C	io.h	低层 I/O 结构类型和函数原型
C	sys-looking.h	定义 looking 函数的 mode 参数
C	malloc.h	内存动态管理函数原型
C	mem.h	内存操作函数原型
C	memory.h	内存操作函数原型
C	new.h	访问 new 和 newhandler 的操作符
C	process.h	与 spawn 和 exec 有关的结构类型和函数原型
C	share.h	定义 share 函数的参数
C	sys-stat.h	打开和创建文件的定义宏
C	sys-timeb.h	时间函数原型和定义返回结构类型 timeb
C	sys-types.h	时间函数定义结构类型 time-t
C	utime.h	时间函数 utime 原型和返回结构类型 utimebub



续表 4.1

C	values.h	定义与机器无关的重要常量
C	varargs.h	变参数的定义宏
C++	bcd.h	C++ bcd 类、运算符重载和函数原型
C++	complex.h	C++ 复数类、运算符重载和函数原型
C++	constrea.h	C++ 类、运算符重载和函数原型
C++	fstream.h	C++ 控制台操作预定义流和流级函数原型
C++	generic.h	C++ 支持文件的流级函数原型
C++	iomamip.h	C++ 流级 I/O 处理器和参数宏
C++	iostream.h	C++ 基本 I/O 流级函数原型
C++	new.h	访问 new 和 newhandler 运算符
C++	stdiostr.h	C++ 支持 FILE 流级函数原型
C++	strstrea.h	C++ 支持内存字符数组的流级函数原型

## 4.9 分类库函数

本节分类列出库中收入的函数名和它们所在的头文件。关于各库函数的详细功能、用法和例子可详见有关的库函数参考手册。

下边以 Borland C++ 为例分类列出支持 DOS 系统的库函数名。

### 4.9.1 归类函数

isalnum	(ctype.h)	isalpha	(ctype.h)
isascii	(ctype.h)	isctrl	(ctype.h)
isdigit	(ctype.h)	isgraph	(ctype.h)
islower	(ctype.h)	isprint	(ctype.h)
ispunct	(ctype.h)	isspace	(ctype.h)
isupper	(ctype.h)	isxdigit	(ctype.h)

### 4.9.2 转换函数

atof	(stdlib.h)	atoi	(stdlib.h)
atol	(stdlib.h)	ecvt	(stdlib.h)
fcvt	(stdlib.h)	gcvt	(stdlib.h)
itoa	(stdlib.h)	_strtod	(stdlib.h)
strtol	(stdlib.h)	_strtold	(stdlib.h)
ultoa	(stdlib.h)	ltoa	(stdlib.h)
strtoul	(stdlib.h)		
toascii	(ctype.h)	_tolower	(ctype.h)
tolower	(ctype.h)	_toupper	(ctype.h)
(ctype.h)			
_strdate	(time.h)	_strtime	(time.h)

## 4.9.3 目录控制函数

chdir	(dir.h)	findfirst	(dir.h)
findnext	(dir.h)	fnmerge	(dir.h)
fnsplit	(dir.h)	getcurdir	(dir.h)
getcwd	(dir.h)	getdisk	(dir.h)
mkdir	(dir.h)	mktemp	(dir.h)
rmdir	(dir.h)	seachpath	(dir.h)
setdisk	(dir.h)		
_chdrive	(direct.h)	closedir	(direct.h)
_getcwd	(direct.h)	_getdrive	(direct.h)
opendir	(direct.h)	readdir	(direct.h)
rewinddir	(direct.h)		
_dosfindfirst	(dos.h)	_dos_findnext	(dos.h)
_dos_getdiskfree	(dos.h)	_dos_getdrive	(dos.h)
_dos_setdrive	(dos.h)		
_fulpath	(stdlib.h)	_makepath	(stdlib.h)
_seachenv	(stdlib.h)	splitpath	(stdlib.h)

## 4.9.4 诊断函数

assert	(assert.h)	matherr	(math.h)
_matherrl	(math.h)	perror	(errno.h)

## 4.9.5 图形函数

arc	(graphics.h)	bar	(graphics.h)
bar3d	(graphics.h)	circle	(graphics.h)
cleardevice	(graphics.h)	clearviewport	(graphics.h)
closegraph	(graphics.h)	detectgraph	(graphics.h)
drawpoly	(graphics.h)	ellipse	(graphics.h)
filellips	(graphics.h)	filpoly	(graphics.h)
floodfil	(graphics.h)	getarccoords	(graphics.h)
getaspectratio	(graphics.h)	getbkcolor	(graphics.h)
getcolor	(graphics.h)	getdefaultpallet	(graphics.h)
getdrivername	(graphics.h)	getfillpattern	(graphics.h)
getfillsettings	(graphics.h)	getgraphmode	(graphics.h)
getimage	(graphics.h)	getlinesettings	(graphics.h)
getgraphmose	(graphics.h)	getmaxx	(graphics.h)
getmaxy	(graphics.h)	getmodename	(graphics.h)
getmoderange	(graphics.h)	getpalette	(graphics.h)

getpalettesize	(graphics.h)	getpixel	(graphics.h)
gettextsettings	(graphics.h)	getx	(graphics.h)
gety	(graphics.h)	graphdefaults	(graphics.h)
grapherrormsg	(graphics.h)	_graphfreemem	(graphics.h)
_graphgetmem	(graphics.h)	graphresult	(graphics.h)
imagesize	(graphics.h)	initgraph	(graphics.h)
installuserdriver	(graphics.h)	installuserfont	(graphics.h)
line	(graphics.h)	linerel	(graphics.h)
lineto	(graphics.h)	moverel	(graphics.h)
moveto	(graphics.h)	outtext	(graphics.h)
outtextxy	(graphics.h)	pieslice	(graphics.h)
putimage	(graphics.h)	putpixel	(graphics.h)
rectangle	(graphics.h)	registerbgidriver	(graphics.h)
registerbgifont	(graphics.h)	restorcrtmode	(graphics.h)
sector	(graphics.h)	getactivepage	(graphics.h)
getallpalette	(graphics.h)	setaspectradio	(graphics.h)
setbkcolor	(graphics.h)	setcolor	(graphics.h)
setcursortype	(graphics.h)	setfillpattern	(graphics.h)
setfillstyle	(graphics.h)	setgraphbufsize	(graphics.h)
setgraphmode	(graphics.h)	setlinestyle	(graphics.h)
setpellete	(graphics.h)	setrgbpellete	(graphics.h)
settextjustify	(graphics.h)	settextstyle	(graphics.h)
setusercharsize	(graphics.h)	setviewport	(graphics.h)
setvisalpage	(graphics.h)	setwritemode	(graphics.h)
setheight	(graphics.h)	setwidth	(graphics.h)

#### 4.9.6 内部函数

用户使用 #pragma intrinsic 或加优化编译时自动选用。

strcpy	(string.h)	strcap	(string.h)
strcmp	(string.h)	strcpy	(string.h)
strncmp	(string.h)	strncpy	(string.h)
strnset	(string.h)	strset	(string.h)
memchar	(mem.h)	mecmp	(mem.h)
memcpy	(mem.h)		
_roll	(stdlib.h)	_rolr	(stdlib.h)
fabs	(math.h)		

#### 4.9.7 输入输出函数

access	(io.h)	_chmode	(io.h)
--------	--------	---------	--------

chmod	(io.h)	chsize	(io.h)
_close	(io.h)	close	(io.h)
creat	(io.h)	creatnew	(io.h)
creattemp	(io.h)	dup	(io.h)
dup2	(io.h)	eof	(io.h)
filelength	(io.h)	ioctl	(io.h)
getftime	(io.h)	lock	(io.h)
locking	(io.h)	lseek	(io.h)
_open	(io.h)	open	(io.h)
isatty	(io.h)	_read	(io.h)
read	(io.h)	setftime	(io.h)
setmode	(io.h)	sopen	(io.h)
umask	(io.h)	tell	(io.h)
unlock	(io.h)	vsscanf	(io.h)
_write	(io.h)		
clearerr	(stdio.h)	fclose	(stdio.h)
fcloseall	(stdio.h)	fdopen	(stdio.h)
feof	(stdio.h)	ferror	(stdio.h)
fflush	(stdio.h)	fgetc	(stdio.h)
fgetchar	(stdio.h)	fgetpos	(stdio.h)
fgets	(stdio.h)	fileno	(stdio.h)
flushall	(stdio.h)	fopen	(stdio.h)
fprintf	(stdio.h)	fputc	(stdio.h)
fputc	(stdio.h)	fputf	(stdio.h)
fread	(stdio.h)	freopen	(stdio.h)
fscanf	(stdio.h)	fseek	(stdio.h)
fsetpos	(stdio.h)	_filesopen	(stdio.h)
ftell	(stdio.h)	fwrite	(stdio.h)
getc	(stdio.h)	getchar	(stdio.h)
gets	(stdio.h)	getw	(stdio.h)
perror	(stdio.h)	printf	(stdio.h)
putc	(stdio.h)	puchar	(stdio.h)
puts	(stdio.h)	putw	(stdio.h)
remove	(stdio.h)	rename	(stdio.h)
rewind	(stdio.h)	rmtmp	(stdio.h)
scanf	(stdio.h)	setbuf	(stdio.h)
setvbuf	(stdio.h)	sprintf	(stdio.h)
sscanf	(stdio.h)	strerror	(stdio.h)(string.h)
strerror	(stdio.h)	tempnam	(stdio.h)

tmpjfile	(stdio.h)	tmpnam	(stdio.h)
ungetc	(stdio.h)	utime	(utime.h)
vfprintf	(stdio.h)	vfscanf	(stdio.h)
vprintf	(stdio.h)	vscanf	(stdio.h)
vsprintf	(stdio.h)		
cgets	(conio.h)	cprintf	(conio.h)
cputs	(conio.h)	cscanf	(conio.h)
getch	(conio.h)	getche	(conio.h)
setcursorype	(conio.h)	ungetch	(conio.h)
getpass	(conio.h)	kbhit	(conio.h)
putch	(conio.h)		
_dos_create	(dos.h)	_dos_creatnew	(dos.h)
_dos_getfileattr	(dos.h)	_dos_gettime	(dos.h)
_dos_open	(dos.h)	_dos_read	(dos.h)
_dos_setfattr	(dos.h)	_dos_setftime	(dos.h)
_dos_write	(dos.h)		
fstat	(sys \ stat.h)	stat	(sys \ stat.h)

#### 4.9.8 各类接口函数 (dos, bios, 8086)

absread	(dos.h)	abswrite	(dos.h)
bdos	(dos.h)	bdosptr	(dos.h)
_chain_intr	(dos.h)	country	(dos.h)
ctrlbrk	(dos.h)	_disable	(dos.h)
disable	(dos.h)	dosexterr	(dos.h)
_dos_getvect	(dos.h)	_dos_keep	(dos.h)
_dos_setvect	(dos.h)	_enable	(dos.h)
enable	(dos.h)	FP_OFF	(dos.h)
FP_SEG	(dos.h)	freemem	(dos.h)
genintrupt	(dos.h)	getcbrk	(dos.h)
getdfree	(dos.h)	getdta	(dos.h)
getfat	(dos.h)	getfatd	(dos.h)
getpsp	(dos.h)	getverify	(dos.h)
getvect	(dos.h)	_harder	(dos.h)
harder	(dos.h)	_hardresume	(dos.h)
hardresume	(dos.h)	_hardretn	(dos.h)
hardretn	(dos.h)	inport	(dos.h)
inportb	(dos.h)	int86	(dos.h)
int86x	(dos.h)	indos	(dos.h)
intdosx	(dos.h)	intr	(dos.h)

keep	(dos.h)	MK_FP	(dos.h)
outport	(dos.h)	outportb	(dos.h)
parsfnm	(dos.h)	peek	(dos.h)
peekb	(dos.h)	poke	(dos.h)
pokeb	(dos.h)	randbrd	(dos.h)
randbwr	(dos.h)	segread	(dos.h)
setcbrk	(dos.h)	setdta	(dos.h)
setvect	(dos.h)	setverify	(dos.h)
sleep	(dos.h)	unlink	(dos.h)
bioscom	(bios.h)	bios_disk	(bios.h)
biosdisk	(bios.h)	_bios_equip	(bios.h)
biosequip	(bios.h)	_bios_keybed	(bios.h)
bioskey	(bios.h)	biosmemory	(bios.h)
biosprint	(bios.h)	_bios_printer	(bios.h)
_bios_serialcom	(bios.h)	biostime	(bios.h)
inp	(conio.h)	inpw	(conio.h)
outp	(conio.h)	outpw	(conio.h)

#### 4.9.9 串与内存块操作函数

memchr	(string.h)(mem.h)	memcmp	(string.h)(mem.h)
memcpy	(string.h)(mem.h)	memccpy	(string.h)(mem.h)
memicmp	(string.h)(mem.h)	memmove	(string.h)(mem.h)
memset	(string.h)(mem.h)	movedata	(string.h)(mem.h)
movemem	(string.h)(mem.h)	setmem	(string.h)
stpcpy	(string.h)	strcat	(string.h)
strchr	(string.h)	strcmp	(string.h)
strcoll	(string.h)	strcpy	(string.h)
strcspn	(string.h)	strdup	(string.h)
strerror	(string.h)	stricmp	(string.h)
strcmpi	(string.h)	strlen	(string.h)
strlwr	(string.h)	strncat	(string.h)
strncmp	(string.h)	strncmpi	(string.h)
strncpy	(string.h)	strnicmp	(string.h)
strnset	(string.h)	strpbrk	(string.h)
strrchr	(string.h)	strrev	(string.h)
strset	(string.h)	strspn	(string.h)
strtok	(string.h)	strxfrm	(string.h)
mblen	(stdlib.h)	mbstowcs	(stdlib.h)
mbtowc	(stdlib.h)	wctomb	(stdlib.h)



## 4.9.10 数学函数

acos	(math.h)(complex.h)	acosl	(math.h)
asin	(math.h)(complex.h)	asinl	(math.h)
atan	(math.h)(complex.h)	atanl	(math.h)
atan2	(math.h)(complex.h)	atan2l	(math.h)
atof	(math.h)	_atold	(math.h)
cabs	(math.h)	cabsl	(math.h)
ceil	(math.h)	ceil	(math.h)
cos	(math.h)	cosl	(math.h)(complex.h)
cosh	(math.h)	coshl	(math.h)(complex.h)
div	(math.h)	exp	(math.h)(complex.h)
expl	(math.h)	fabs	(math.h)
fabsl	(math.h)	fcvt	(math.h)
floor	(math.h)	floorl	(math.h)
fmode	(math.h)	fmodel	(math.h)
frexp	(math.h)	frexp	(math.h)
hypot	(math.h)	hypotl	(math.h)
ldexp	(math.h)	ldexpl	(math.h)
ldiv	(math.h)	log	(math.h)(complex.h)
logl	(math.h)	log10	(math.h)(complex.h)
log10l	(math.h)	matherr	(math.h)
_matherrl	(math.h)	modf	(math.h)
modfl	(math.h)	poly	(math.h)
polyl	(math.h)	pow	(math.h)
powl	(math.h)	pow10	(math.h)
pow10l	(math.h)	sin	(math.h)(complex.h)
sinl	(math.h)	sinh	(math.h)(complex.h)
sinhl	(math.h)	sqrt	(math.h)
sqr	(math.h)	tan	(math.h)(complex.h)
tanl	(math.h)	tanh	(math.h)(complex.h)
tanh	(math.h)		
arg	(complex.h)	complex	(complex.h)
image	(complex.h)	norm	(complex.h)
polar	(complex.h)	real	(complex.h)
abs	(complex.h)(stdlib.h)	atoi	(stdlib.h)
atol	(stdlib.h)	ecvt	(stdlib.h)
fcvt	(stdlib.h)	gcvt	(stdlib.h)
itoa	(stdlib.h)	labs	(stdlib.h)

_lrotl	(stdlib.h)	_lrotr	(stdlib.h)
ltoa	(stdlib.h)	rand	(stdlib.h)
random	(stdlib.h)	randomize	(stdlib.h)
_rotl	(stdlib.h)	_rotr	(stdlib.h)
srand	(stdlib.h)	strtod	(stdlib.h)
strtol	(stdlib.h)	_strtold	(stdlib.h)
strtoul	(stdlib.h)	ultoa	(stdlib.h)
clear87	(float.h)	_fpreset	(float.h)
_stat87	(float.h)		

#### 4.9.11 动态内存管理函数

brk	(alloc.h)	calloc	(alloc.h) (stdlib.h)
coreleft	(alloc.h)		(stdlib.h) farcalloc (alloc.h)
farcoreleft	(alloc.h)	farfree	(alloc.h)
farheapcheck	(alloc.h)	farheapcheckfree	(alloc.h)
farheapfillfree	(alloc.h)	farheapwalk	(alloc.h)
farmalloc	(alloc.h)	farrealloc	(alloc.h)
free	(alloc.h) (stdlib.h)	heapcheck	(alloc.h)
heapcheckfree	(alloc.h)	heapchecknode	(alloc.h)
heapwalk	(alloc.h)	malloc	(alloc.h) (stdlib.h)
realloc	(alloc.h) (stdlib.h)	sbrk	(alloc.h)
alloc	(malloc.h)		
allocmem	(dos.h)	_dos_allocmem	(dos.h)
_dos_freemem	(dos.h)	_dos_setblock	(dos.h)
setblock	(dos.h)		

#### 4.9.12 杂项函数

longjmp	(setjmp.h)	setjmp	(setjmp.h)
delay	(dos.h)	nosound	(dos.h)
sound	(dos.h)		
localeconv	(locale.h)		

#### 4.9.13 进程控制函数

abort	(process.h)	_c_exit	(process.h)
_exit	(process.h)	execl	(process.h)
execle	(process.h)	execlp	(process.h)
execlpe	(process.h)	execv	(process.h)
execve	(process.h)	execvp	(process.h)
execvpe	(process.h)	_exit	(process.h)

exit	(process.h)	getpid	(process.h)
spawnl	(process.h)	spawnle	(process.h)
spawnlp	(process.h)	spawnlpe	(process.h)
spawnv	(process.h)	spawnve	(process.h)
spawnvp	(process.h)	spawnvpe	(process.h)
raise	(signal.h)	signal	(signal.h)

#### 4.9.14 窗口文本显示函数

clrcol	(conio.h)	clrscr	(conio.h)
delline	(conio.h)	gettext	(conio.h)
gettextinfo	(conio.h)	gotoxy	(conio.h)
highvideo	(conio.h)	insline	(conio.h)
lowvideo	(conio.h)	movetext	(conio.h)
normvideo	(conio.h)	puttext	(conio.h)
setcursortype	(conio.h)	textattr	(conio.h)
textbackground	(conio.h)	textcolor	(conio.h)
texymode	(conio.h)	wherex	(conio.h)
wherey	(conio.h)	window	(conio.h)

#### 4.9.15 日期时间函数

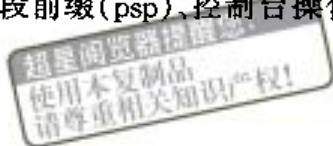
asctime	(time.h)	ctime	(time.h)
difftime	(time.h)	ftime	(sys \ time.h)
gmtime	(time.h)	localtime	(time.h)
mktime	(time.h)	stime	(time.h)
strftime	(time.h)	time	(time.h)
tzset	(time.h)		
_dos_getdate	(dos.h)	_dos_gettime	(dos.h)
_dos_setdate	(dos.h)	_dos_settime	(dos.h)
dostounix	(dos.h)	getdate	(dos.h)
gettime	(dos.h)	settime	(dos.h)
setsate	(dos.h)	unixtodos	(dos.h)
_bios_timeofday	(bios.h)		

#### 4.9.16 变参数表函数

va_arg	(stdarg.h)	va_end	(stdarg.h)
va_start	(stdarg.h)		

## 4.10 全局变量

全局变量是提供给库函数和用户使用的重要变量。高级程序员需要掌握全局变量。初学者短时可能不会感到有这种需要,暂时可以放弃。下面以 Borland C 为例进行介绍。它的全局变量涉及到环境变量、命令行参数、堆栈深度、近堆长度、当前程序段前缀(psp)、控制台操作方式、时区、错误信息等广泛的方面。



### 4.10.1 argc

**功能:** 保存命令行参数字符串的串数。

**用法:** `extern int _argc;`

**定义在:** `dos.h`

**解释:** 保存程序启动时传给 `main()` 的输入参数字符串的串数。

### 4.10.2 argv

**功能:** 命令行参数的字符指针数组。

**用法:** `extern char *_argv[ ];`

**定义在:** `dos.h`

**解释:** 该数组保存程序启动时传给 `main()` 的各参数的字符指针。

### 4.10.3 ctype

**功能:** 保存 ASCII 字符属性的数组。

**用法:** `extern char _ctype[ ];`

**定义在:** `ctype.h`

**解释:** 该数组以 ASCII 值 + 1 为索引,各元素为相应 ASCII 字符的属性字节,用于 `isdigit` 和 `isprint` 等函数。

### 4.10.4 daylight

**功能:** 夏令时标志字。

**用法:** `extern int daylight;`

**定义在:** `time.h`

**解释:** 由 `tzset()`, `ftime()` 和 `localtime()` 等函数将其置 1 (设为夏令时) 或清 0 (设为标准时间)。

### 4.10.5 directvideo

**功能:** 视频 RAM 标志字。

**用法:** `extern int directvideo;`

**定义在:** `conio.h`

**解释:** 当 `directvideo` 置为 1 时,应用程序向控制台的输出直接送视频 RAM。但是,要

求视频卡提供视频 RAM 硬件。directvideo 置 0 时,则控制台的输出是通过 ROM BIOS 的调用来实现的,显示速度较慢。

#### 4.10.6 environ

**功 能:** 存放 DOS 环境变量的字符指针数组。

**用 法:** `extern int * environ[ ];`

**定义在:** `dos.h`

**解 释:** 环境变量由下式设置:

`envvar = 环境变量字符串`

其中:envvar 为环境变量名(如 PATH 等)。每个程序开始执行时,DOS 将当时的所有的环境变量字符串自动传给它。environ 字符指针数组的项数由 `putenv( )` 惟一修改。读取环境变量字符串用 `getenv( )`。

#### 4.10.7 error, \_doserrno, sys\_errlist, sys\_nerr

**功 能:** 提供 DOS 系统运行出现错误或库例程出现错误的信息。

**用 法:** `extern int errno, _doserrno, sys_errno;`

`extern char * sys_errlist[ ];`

**定义在:** `error.h, stdio.h, dos.h(对 _doserrno)`

**解 释:** `_doserrno` 存放 DOS 系统运行错误码。`errno` 存放库例程的错误码和 DOS 由 UNIX 继承来的系统运行错误码。`sys_errlist` 是存放多个错误信息串的字符指针数组。`sys_nerr` 是 `sys_errlist` 的项数。除去 `_doserrno` 之外的 `error`, `sys_errlist` 和 `sys_nerr` 均服务于错误信息函数的打印。关于 `_doserrno` 的系统运行错误码的打印处理,请见 DOS 参考手册。关于 `sys_errlist` 的错误信息助记符的含义见表 4.2 所示。

表 4.2 `sys_errlist` 的错误信息助记符及其含义

助记符	含 义
E2BIG	参数表太长
EACCES	权限错
EBADF	非法文件
ECONTR	内存块破坏
ECURDIR	企图删当前目录
EDOM	域错误
EEXIST	文件已经存在
EFAULT	未知错误
EINVACC	无效存取
EINVAL	无效变量
EINVDAT	无效数据
EINVDRV	无效驱动器号
EINVENV	无效环境串
EINVFMT	无效格式
EINVFNC	无效功能号



续表 4.2

助记符	含 义
EINVMEM	无效内存块地址
EMFILE	打开文件太多
ENMFILE	没有更多文件
ENODEV	无此设备
ENOENT	无此文件或目录
ENOEXEC	运行格式错
ENOFIL	无此文件或目录
ENOMEM	无足够内存
ENOPATH	无此路径
ENOTSAM	无相同设备
ERANGE	结果超范围
EXDEV	设备交叉连结
EZERO	0 除错

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

## [例 4.4]

```
#include <errno.h>
#include <stdio.h>
extern char *sys_errlist[];

int main()
{
    int i=0;
    while(sys_errlist[i++] )
        printf("%s\n", sys_errlist[i]);
    return 0;
}
```

## 4.10.8 fmode

**功 能：** 决定文件的缺省打开和传送的方式。

**用 法：** extern int \_fmode;

**定义在：** fcntl.h

**解 释：** 文件的打开和传送方式有两种：文本方式和二进制方式。在打开文件时需要指定。因为设定了文件的缺省方式，按缺省方式打开文件时，可以不再指定。在调用 fopen(), fdopen(), freopen() 时还可以最后指定文件的打开方式。而且，这时候的方式暂时覆盖缺省方式。两种方式的不同在于：文本方式在读的时候将回车和换行变为单换行，而在写的时候将单换行再换回去；对于二进制方式没有这种转换。

## 4.10.9 heaplen

**功 能：** 存放近堆的长度。

**用 法：** extern unsigned \_heaplen;

**定义在：** dos.h

**解 释：** heaplen 只用于小数据模式(tiny, small, medium)。大数据模式(compact, large,



huge)用不着。heaplen 给小数据模式指定堆的长度。对于 tiny 存储模式 heaplen 的最大值不应超过:

$$64K - \text{code} - \text{psp}(256\text{byte}) - \text{globle data} - \text{stack}(\text{byte})$$

对于 small 和 medium 存储模式, heaplen 最大值不应超过:

$$64K - \text{globle data} - \text{stack}(\text{byte})$$

#### 4.10.10 \_new\_handler

功 能: 存放管理错误函数的地址。

用 法: typedef void (\* pvf)();

pvf \_new\_handler;

定义在:

解 释: 缺省的情况下, 运算符 new 的分配错只转向简单地终止用户程序的函数。若想使用用户自定义的管理函数时, 应将自定义函数的地址放入 \_new\_handler。

#### 4.10.11 \_osmajor, \_osminor

功 能: 分别存放 DOS 版本的主号和付号。

用 法: extern unsigned char \_osmajor, \_osminor;

定义在: dos.h

解 释: 如果 DOS 版本号为 6.22, 则主号是 6 和副号是 22。分别存放到 \_osmajor 和 \_osminor 中。

#### 4.10.12 \_ovrbuffer

功 能: 存放覆盖缓冲区的长度。

用 法: extern unsigned \_ovrbuffer = size;

定义在: dos.h

解 释: 可覆盖缓冲区的缺省长度, 应是最长实需缓冲区的 2 倍。如果内存交换过频, 应加大 \_ovrbuffer 的值。

#### 4.10.13 \_psp

功 能: 存放当前 \_psp 的段地址。

用 法: extern unsigned \_psp;

定义在: dos.h

解 释: \_psp 是当前进程的描述段。

#### 4.10.14 \_stklen

功 能: 存放堆栈的深度。

用 法: extern unsigned \_stklen;

定义在: dos.h

解 释: 堆栈深度最小为 128 个字节(自动设置), 缺省为 4 KB。用户可以向 \_stklen 赋



值来指定堆栈深度。下面给出各种存储模式堆栈深度的最大值。

tiny: 64K - 256 - code - globle data - heap  
 small, medium: 64K - globle data - heap  
 compact, large: 64K - globle data  
 huge: 64K

[例 4.5] #include <stdio.h>

```
extern unsigned _stack = 54321u;
void main( )
{ printf("the stack length is: %u\n", _stack); }
```

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

#### 4.10.15 timezone

功 能：存放当地时间与格林威治时间(GMT)相差的秒数。

用 法：extern long timezone;

定义在：time.h

解 释：本变量由 tzset( ) 求出, 供时间、日期函数使用。

#### 4.10.16 tzname

功 能：提供时区名和夏令时区名的字符指针数组。

用 法：extern char \* tzname[ 2];

定义在：time.h

解 释：tzname[ 0] 和 tzname[ 1] 分别指向 TZ 环境集中的时区名和夏令时区名。没有夏令时区名的指针 tzname[ 1] 为 null。

#### 4.10.17 \_version

功 能：存放 DOS 的版本号。

用 法：extern unsigned \_version;

定义在：dos.h

解 释：DOS 版本的主号放低字节, 副号放高字节。

#### 4.10.18 \_wscroll

功 能：控制台 I/O 函数滚屏标志字。

用 法：extern int \_wscroll;

定义在：dos.h

解 释：\_wscroll 置 1 是滚屏; 置 0 不滚屏。缺省时滚屏。

#### 4.10.19 \_8087

功 能：协处理器标志字。

用 法：extern int \_8087;

定义在：dos.h

**解 释：** 引导程序自动检测硬件协处理器\_8087 的存在。置 1 是有硬件协处理器存在；不存在置 0。存在 8087 置 1；80287 置 2；80387 置 3。可以用 SET 命令将环境变量 87 置为 YES 或 NO(如 SET87 = YES/NO)。SET 命令影响\_8087。

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

## 第五章 预处理器

### 5.1 前言

过去编译器执行两遍编译。第一遍主要是把包含文件替换进来,进行宏扩展、测试条件编译等,虽然生成了中间结果,但被随后执行的第二遍编译所废弃了。现在,在编译前先用独立的预处理器把源文件处理一遍,内容与第一遍编译相当,但是产生的中间结果却被用文件保留下来。再把经过预处理后的源文件交给编译器。所以,现在的编译只执行一遍。

预处理器为用户带来的好处有:

1. 把包含文件的正文替换进来,如标准头文件和自编头文件,其内容包括符号常量、复合变量原型、用户定义的变量类型原型和函数的原型说明等。
2. 对定义宏进行宏扩展,减少了编程量,改进源程序的可读性,参数宏更减少了函数调用的开销。
3. 条件编译改善了编程的灵活性,改善了可移植性。

下面不是介绍预处理器本身,而是介绍编写源文件时,如何加入预处理器伪指令。它们是专门用于指示预处理器怎样进行预处理,部分是指示编译器如何编译。对于后者,有些功能与编译命令中所给编译参数的功能相同,只不过提前在编写源文件时就给定了。其中,有的控制是在编译命令中无法作精细控制的,则只能在源文件中用预处理器伪指令予以设定。

所有的预处理器伪指令行都以#号打头,以区别于源文件中的说明行与语句行。

下面分别介绍各种预处理伪指令。

### 5.2 包含文件伪指令

格式:

```
#include <头文件名.h>
#include "头文件名.h"
#include 宏标识符
```

其中:

- (1) 习惯上头文件名后用.h作为扩展名,可以带或不带路径。
- (2) 标准头文件与自定义头文件。

·尖括号对内的头文件为标准头文件。标准头文件按 dos 系统的环境变量 include 所指定的目录顺序搜索头文件。

·双引号对内的头文件名为用户自定义头文件。搜索时,首先在当前目录(通常为源文件所在目录)中搜索,其次按环境变量 include 指定的目录顺序搜索。

·搜索到头文件后,就将该伪指令就地用头文件全内容替换之。

(3) 第三种格式中的宏标识符预处理器首先对它进行宏扩展。宏扩展后得到的可能是尖括号内的头文件名或双引号对内的头文件名。其后,就可以按前二种格式之一来处理。

[例 5.1]

```
#include <stdio.h>
```

是准标头文件,按环境变量 include 指定的目录顺序搜索 stdio.h。

[例 5.2] 定义宏标识符。

```
#define MYINCLUDE "C:\borlandc\include\MY.h"
```

有

```
#include MYINCLUDE
```

经宏扩展后为

```
#include "C:\borlandc\include\MY.h"
```

它是自定义头文件,首先搜索当前目录。



## 5.3 伪指令宏

伪指令宏分为:

- 简单宏
- 参数宏
- 条件宏
- 预定义宏
- 宏释放

### 5.3.1 简单宏

定义格式:

```
#define 宏标识符 宏体
```

其中:

(1) 宏体是由单词序列组成。宏体超长时,允许使用续行符“\”进行续行,续行符和其后的换行符 \n 都不会进入宏体中。

(2) 在定义宏时,应尽量避免使用 C 语言的关键字和预处理器的预定义宏,以免引起灾难性的后果。

(3) 宏的用法:在源文件中,用预处理器伪指令定义过宏标识符之后,就可以用宏标识符编写程序。当源文件被预处理器处理时,每遇到该宏标识符,预处理器便在宏的所在处将宏扩展为宏体。

[例 5.3]

```
#define ER "Errorfound!"
```

```
printf(ER);/* 预处理器将 ER 替换为 printf("Errorfound!") */
```

### 5.3.2 参数宏

定义格式:

#define 宏标识符(形式参数表) 宏体

其中:

(1) 形式参数表为逗号分割的形式参数。

(2) 宏体是由单词序列组成。有些单词包含参数表的形式参数。宏体超长时,允许使用续行符“\”进行续行,续行符和其后的换行符\n都不会进入宏体中。

(3) 使用参数宏时,形式参数表应换为同样个数的实参数表,这一点类似于函数的调用。事实上,许多库函数是用参数宏写的。参数宏和函数的区别:一是形式参数表中没有类型说明符;二是参数宏在时空的开销上比函数都要小。

(4) 预处理器在处理参数宏时使用两遍宏展开。第一遍展开宏体,第二遍对展开后的宏体用实参数替换形式参数。

[例 5.4]

```
#define SQR(x, y) sqrt((x)*(x)+(y)*(y))
```

源文件中有:

```
z = SQR(a+b, a-b); /* 替换为 sqrt((a+b)*(a+b)+(a-b)*(a-b)); */
```

### 5.3.3 宏释放

用于释放原定义的宏标识符。经释放后的宏标识符可以再次用于定义其他宏体。宏释放格式:

#undef 宏标识符

[例 5.5]

```
#define BLOCK - SIZE 512
```

...

```
buf = BLOCK - SIZE * blks; /* 宏扩展为 buf = 512 * blks; */
```

...

```
#undef BLOCK - SIZE
```

```
#define BLOCK - SIZE 128
```

...

```
buf = BLOCK - SIZE * blks; /* 宏扩展为 buf = 128 * blks; */
```

### 5.3.4 条件宏定义

先测试是否定义过某宏标识符,然后决定如何处理。格式有二种:

#ifdef 宏标识符

#ifndef 宏标识符

#undef 宏标识符

#define 宏标识符 宏体

#define 宏标识符 宏体

#else

#else

#undef 宏标识符

#define 宏标识符 宏体

#define 宏标识符 宏体

#endif

#endif

其中:

(1) 左边格式是测试存在,右边是测试不存在。

(2) 左边格式和右边格式的#else部分可以没有,有与没有意义不同。



## [例 5.6]

```
#ifdef BLOCK - SIZE
    # undef BLOCK - SIZE
    # define BLOCK - SIZE 128
#endif
```

与

```
#ifndef BLOCK - SIZE
    # define BLOCK - SIZE 128
#else
    # undef BLOCK - SIZE
    # define BLOCK - SIZE 128
#endif
```

意义不同。

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

## 5.3.5 预定义宏

预处理器的预定义宏一般有：

```
__DATE__ __TIME__
__FILE__ __LINE__
__STDC__
```

它们的特征是标识符的前和后都有双下划线。这些预定义宏的宏体分别是当时的日期、时间、所在文件名、所在文件的行数，而\_\_STDC\_\_当与ANSI C相兼容时，它定义的宏体是1。

## 5.3.6 宏体中使用转义符 # 和合并符 ##

## 1. 转义符 #

定义参数宏时，在宏体中允许将 # 号加于参数之前。此时的 # 号为预处理器的转义符。在宏扩展时将 # 后的参数转义为字符串。

## [例 5.7]

```
#define PRINTF(x) printf("# x " = %d \n", x)
```

在源文件中有：

```
PRINTF(salary);
```

经预编译器扩展为：

```
printf("salary " = %d \n", salary);
```

即：printf("salary = %d \n", salary);

## 2. 合并符 ##

在宏体内，合并符 ## 将两边的单词合并为一个单词。合并符两边允许有任意多的空格符。预处理器在处理时，将这些空格符和 ## 符一并删去，使合并符两边的单词合而为一。

## [例 5.8]

```
#define printconv(x) printf("token" # x " = %d", token ## x)
printconv(5); /* 等于 printf("token5 = %d", token5); */
```

## 5.4 条件编译伪指令

本节的伪指令是写给编译器的,指示编译器在满足某一条件时仅编译源文件中与之相应的部分。预处理器对它的作用仅是扫描其中的宏并进行宏扩展,其他内容不动,留给编译器对它进行处理。

条件编译格式如下:

```
# if (条件表达式 1)
...
# elif (条件表达式 2)
...
# elif (条件表达式 n)
...
# else
...
# endif
```

其中:

- (1) 条件表达式允许使用宏标识符。
- (2) 编译时,编译器仅对 `#if()` ..... `#endif` 之间,满足某一条件表达式的源文件部分进行编译。

## 5.5 #pragma 伪指令

格式:

`#pragma` 编译器伪指令

其中:

- (1) 编译器伪指令——C 语言编译器所用的伪指令(注意:不是预处理器伪指令)。
- (2) `#pragma` 伪指令的用途——与编译命令行中作为命令参数的控制选项一样。只不过它提前在源程序中给出。编译器的控制选项中有一些是只能在源程序中用 `#pragma` 伪指令格式给出的。所以, `#pragma` 伪指令格式比编译器的命令行参数要细致一些。

## 5.6 #line 伪指令

在源文件中存在插入的包含文件(用 `#include` 得到)。当编译有错时或在交叉引用时,希望使用原来的文件名和原来的行号,则应事先在源文件中使用 `#line` 伪指令加以登记。格式如下:

`#line` 整型常量 ["文件名"]

其中:

- (1) 整型常量是原所在文件中下一行的行号(该文件现已被插入在别的文件中)。

(2) “文件名”是下一行的原所在文件的文件名(现已被插入在别的文件中)。文件名可以缺省,缺省时是当前文件名。

[例 5.9] 在 temp.c 中使用 #line 伪指令。

```
/* temp.c */
#include <stdio.h>
#line 4 "junk.c"
void main()
{
    printf("in line %d of %s", __LINE__, __FILE__);
#line 12 "temp.c"
    printf("\n");
    printf("in line %d of %s", __LINE__, __FILE__);
#line 8
    printf("\n");
    printf("in line %d of %s", __LINE__, __FILE__);
}
```

经编译后为:

```
temp.c 1:/* temp.c */
temp.c 2:
c:\borland\include\stdio.h 1:
c:\borland\include\stdio.h 2:
...
c:\borland\include\stdio.h n:
temp.c 3:
junk.c 4: void main()
junk.c 5: {
junk.c 6:     printf("in line %d of %s", 6, "junk.c");
junk.c 7:
temp.c 12:     printf("\n");
temp.c 13:     printf("in line %d of %s", 13, "temp.c");
temp.c 14:
temp.c 8:     printf("\n");
temp.c 9:     printf("in line %d of %s", 9, "temp.c");
}
```

经执行后显示出:

```
in line 6 of junk.c
in line 13 of temp.c
in line 9 of temp.c
```

## 5.7 #error 伪指令

本伪指令的作用是,嵌入在条件编译语句中,用以捕捉不可预料的编译条件。捕捉到时,用 #error 伪指令提供出错信息并停止编译。它的格式如下:

```
#error errmsg
```

其中:errmsg 是程序员规定的错误信息。有错时,输出如下的格式:

```
Error: 文件名 line #: Error directive:errmsg
```

[例 5.10] 已知 MYVAL 只能是 0 或 1。有意在正常不会出现的编译条件语句部分中插入 #error 伪指令语句,等待捕捉意外。当意外出现时,提供信息并停止编译。有:

```
#if ( MYVAL != 0 && MYVAL != 1 )
```

```
#error MYVAL must be defined to either 0 or 1
```

编译器执行条件编译语句,当 MYVAL 不是 0 也不是 1 时,给出下面的信息后停止编译:

```
Error: temp.c line 60: Error directive: MYVAL must be defined to either 0 or 1
```

# 第二部分

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

## C51 (8051 用 8 位嵌入式 C 语言)

### 第六章 C51 前言

微控制器上使用的 C 都是直接针对具体硬件的。目前,各生产微控制器的厂家,已经重视为自己生产的微控制器配备 C 语言。这已经成为趋势。可以说,掌握了 C 语言,也就打开了轻松使用各个厂家微控制器的大门,所需做的仅仅是完成 C 程序在微控制器之间的移植问题。软件在微控制器之间通用化的日子已经指日可待了。

任何一家公司开发的 C 语言,都必须符合 ANSI C 的标准。所以,不管哪一家的 C,其主要部分定会保持一致性,不同的只是非 ANSI C 的扩展部分。异种机之间的 C 尚可移植,同是 8051 的 C,差异就更小了。目前,不同软件公司对同一个微控制器配备的 C 都会存在小的区别。本书主要介绍当前公认效率较高的 Franklin C51,以它为蓝本。有了一家 C51 的基础,再转入其他公司的 C51 也就容易了。所以,集中精力,学好一家公司的 C51 是首要的。

需要特别指出的是,在认识上要重视基础部分关于 80X86 上的 C 的介绍。这不单单是因为它是学习 8051 上 C 的基础,而且学会 80X86 的 C 也为开发 8051 C 提供额外的便利。须知,PC 机是当今 8051 仿真器使用最广泛的寄主机。而且,用于开发 80X86 C 语言程序的集成开发环境(IDE)比开发 8051 C 语言程序的环境应该说还是要完善得多。因此,完全有可能先用 80X86 的编辑器、编译器、连接器、模拟器在 C 语言级上开发 8051 应用程序,直到在模拟器上调试通过;然后,再向 8051 移植;最后,再用 8051 的仿真器调试。

另外需要说明的是,目前微控制器上的 C 语言尚停留在 C 的阶段。C 与 C++ 之间的跨度是很大的,微控制器如何升入 C++ 尚待探讨。新近面世的 16 位 51XA 上的 C 语言,已开始形式上涉及了 C++ 的一些内容。

## 第七章 C51 说明



### 7.1 C51 简单变量说明

格式:

[存储类说明符③] 类型说明符① [修饰符④] 标识符② [= 初值⑤] [, 标识符 [= 初值⑤]] ...;

③	①	⑤	②
<div>auto extern static register</div>	<div>unsigned char char unsigned int int unsigned long long float bit sfr sfr16 sbit</div>	<div>data bdata idata pdata xdata code</div>	<div>{ 变量名 [= 初值 ] * 指针名 [= &amp; 变量名] }</div>

其中:

(1) 类型说明符①和标识符②两部分必须存在。

(2) C51 类型说明符部分与基础部分不同之处有:

• 整型数类型与基础部分虽是一样的, 但应指出, 算术运算要尽可能的使用无符号数 (unsigned char, unsigned (int) 和 unsigned long)。因为, 8051 的指令集只支持无符号数算术运算。有符号数的算术运算在 C51 中是用函数实现的, 效率较低, 非必要时不要使用。

• 浮点数只支持到 float(32 位), 没有 double 和 long double。

• 增加了 bit(位)类型、sfr(预定义特殊功能寄存器)类型、sfr16(预定义 16 位特殊功能寄存器)类型、sbit(预定义可位寻址特殊功能寄存器)类型。

(3) 存储类说明符③、初值说明符⑤和多变量说明⑤等部分与基础部分相同, 不再叙述。

(4) 修饰符④部分, C51 有新的内容。针对 8051 存储空间的多样性提出了修饰存储空间的修饰符, 用于指明所定义的变量应分配在什么样的存储空间。修饰存储空间的修饰符有:

• data——片内直接寻址 RAM 空间(寻址范围为 0~127)。存取速度最快。

• idata——片内间接寻址 RAM 空间(寻址范围为 0~255)。

• pdata——片外 RAM 第一页空间(寻址范围为 0~255)。存取使用 MOVX @RI 间接



寻址。有的 8051 派生芯片将这部分空间移入片内,有利于提高存取速度。

·xdata——片外 RAM 空间(寻址范围为 0~65 535)。存取使用 MOVX @DPTR 间接寻址。

·code——片内外统一编址的 ROM 空间(寻址范围为 0~65 535)。片外 ROM 部分存取使用 MOVC @A+DPTR 间接寻址。

·bdata——片内位寻址 RAM 空间(片内 RAM 0x20~0x2F 空间,共占 16 个字节。它们的位地址范围为 0~127)。本空间允许按位或按字节直接寻址。

(5) 修饰符部分是可有可无的,如果变量和指针未指定修饰符部分,那么它们究竟定位在哪个存储空间呢?在这种修饰符缺省的情况下,定位空间决定于编译时选用的存储模式。

(6) 应该特别强调指出,在 C51 中习惯上把修饰符部分放在类型说明符之前。

(7) 定义性说明与引用性说明的规则与基础部分相同。定义性说明分配内存,引用性说明不分配内存。

(8) 在本格式中凡与基础部分不同的项均用下划线标出。

### 7.1.1 类型说明符 bit

bit 是 C51 特有的类型说明符。它有如下特点:

·bit 类型可用的修饰符——最正规的应是 bdata,但是用了 data 和 idata 也不为错,其他修饰符不可用。

[例 7.1]

```
bdata bit display-flag;    /* 正确 */
data bit display-flag;     /* 正确 */
idata bit display-flag;    /* 正确 */
bit display-flag;          /* 正确 */
pdata bit display-flag;    /* 错 */
static bit direction-bit;  /* 正确 */
extern bit lock-port1;     /* 正确 */
```

·类型不能说明为指针和数组,如:

```
bit *bptr;                 /* 错 */
bit b-array[3];            /* 错 */
```

·位变量可用于结构和联合。

·位变量可以作为函数的参数和返回量。用寄存器组修饰的函数不能用位变量做返回量。因为按 C51 的规定,返回的位变量是放在 ACC 的进位位中返回的。而用寄存器组修饰的函数,函数中所用的寄存器组在返回之前改变了,使寄存器组中返回的量不正确。

### 7.1.2 预定义特殊功能寄存器说明符 sfr 和 sfr16

微控制器有许多特殊功能寄存器。它们在片内 RAM 空间中已经安排了绝对地址。并且,也为它们用预定义标识符起了名字。C51 要做的工作并不是再给它们定义新的标识符,而是承认预定义的标识符并把它与对应的绝对地址联系起来。格式如下:

```
sfr    特殊功能寄存器预定义标识符 = 绝对地址;
sfr16  特殊功能寄存器预定义标识符 = 绝对地址;
```

·可以与变量标识符一样,用预定义标识符去存取特殊功能寄存器。

·定义 sfr 类型和 sfr16 类型的变量时,标识符不可自行选用,而是必须使用预定义标识符。而且,也必须把原来分配好的绝对地址赋给预定义标识符。

[例 7.2]

```
sfr p0 = 0x80;
```

```
sfr p1 = 0x90;
```

·特殊功能寄存器一般都是按 8 位存取的,但也有按 16 位存取的。按 16 位存取的需要用 sfr16 类型来说明预定义标识符。

[例 7.3]

```
sfr16 T2 = 0xCC; /* T2 的低地址 T2L = 0xCC, 高地址 T2H = 0xCD */
```

```
sfr16 RCAP2 = 0xCA; /* RCAP2L = 0xCA, RCAP2H = 0xCB */
```

例中的特殊功能寄存器 T2 被说明后,就可以像 int 或 unsigned 类型的变量一样,用赋值语句为它赋值。

[例 7.4]

```
T2 = 0x1234;
```

其结果:T2L 为 0x34, T2H 为 0x12。

### 7.1.3 预定义特殊功能寄存器位说明符 sbit

格式:

sbit 预定义 SFR 位标识符 = 可按位寻址的预定义 SFR 标识符 ^ 常量;

sbit 预定义 SFR 位标识符 = 可按位寻址的预定义 SFR 的绝对地址 ^ 常量;

sbit 预定义 SFR 位标识符 = SFR 的绝对位地址。

其中:

可位寻址的 SFR 分布在内部 RAM 的 0x80~0xFF 空间可被 8 整除的地址上。

[例 7.5]

```
sfr PSW = 0xD0;
```

```
sbit CY = PSW ^ 2;
```

```
sbit CY = 0xD0 ^ 2;
```

```
sbit CY = 0xD7;
```

经过 sbit 说明过的 CY 可以当做位变量名进行存取。

### 7.1.4 在 bdata RAM 空间定义位变量(借用位类型符 sbit)

给 bdata 存储空间的位变量作定义性说明有两种方法:

·直接用 bit 类型说明符说明为位变量(前面已经叙述)。

·先说明为 bdata 存储空间的 char 或 int 变量,再进一步把 char 或 int 变量用 sbit 位类型符说明成位变量。

下面是后一种方法的举例:

[例 7.6]

```
bdata int ivar;
```



```
sbit ivarbit10 = ivar ^ 10;
```

## [例 7.7]

```
bdata char myarray[3];
sbit  myarray 21 = myarray[2] ^ 1;
sbit  myarray 32 = myarray[3] ^ 2;
```

## [例 7.8] 判浮点数的符号位是否为正。

C51 中浮点数要按 IEEE-754 标准的规则存放在内存中。下面是浮点数在内存中存放的方法：

	7	0
高地址	S	E
	E	M
	M	
低地址	M	

其中：S—符号位(1 位)(第 31 位)。

E—偏移 127 后的阶码(8 位)(第 23~30 位)。

M—尾数(23 位)(第 0~22 位)。

为找出浮点数的符号位,最简单的方法是利用联合(union),如:

```
union float long
{
    float bdata f;
    long bdata l;
} fl;
sbit float _sign = fl.l ^ 31;
if (!float _sign) /* 符号位为正 */
{ ... }
else
{ ... }
```

## 7.2 C51 复合变量说明

复合变量包括数组、结构和联合。它们的说明规则与基础部分一样,所以不再重复。

## 7.3 C51 指针变量说明

C51 的存储器空间多样且大小不一,存取效率差异也大,致使指针变量在说明时变得复杂。考虑诸多方面,说明指针变量有如下格式:

[存储类说明符①] 类型说明符② [修饰符③] [指针定位修饰符④] 标识符⑤ [= 地址初值⑥] [, 标识符 [= 地址初值] ⑦] ...;

③	①	①	⑦	②
static auto register extern	unsigned char char unsigned int int unsigned long long float struct union enum void	data idata pdata xdata code bdata 缺省 时用 三字 节通 用指 针	data idata pdata xdata code 缺省时 决定于 编译用 存储模 式	* 标识符 (* 标识符)[ ] (* 标识符)[ ][常量表达式] ** 标识符 { data idata } * )0x8 位绝对地址 pdata { xdata code } * )0x16 位绝对地址

其中:

(1) 第① ② ③部分与7.1节一样。其他:

·初值部分③——在此应改为地址初值。

·修饰部分①——为指针指向的对象定位存储空间。定位后,有助于肯定指针的长度。

data, idata, pdata 空间 一字节指针

xdata, code 空间 二字节指针

未指定空间 三字节通用指针(通用指针见后)

·指针定位修饰符⑦部分——给指针本身定位所在存储空间。一般不预指定,只在必要时才预指定。不指定时,由编译时所用存储模式决定。

·标识符部分②——标识符部分是必需的。标识符有下列可能:

* 标识符	标识符是指针名
(* 标识符)[ ]	标识符是数组指针
(* 标识符)[ ][常量标识符]	标识符是二维数组指针
** 标识符	标识符是二重指针
(data idata pdata * )8 位绝对地址	抽象指针
(data code * )16 位绝对地址	抽象指针

(2) 给指针作定义说明时,除必须说明所指对象的类型外,经常还要指定对象所在的存储器空间,以便及时确定指针的长度。此外,指针本身也有个放在哪个存储器空间的问题。指针本身放在片外存储空间时取指针的时间要长。所以,时常希望把常用指针定位到片内存储空间。

### 7.3.1 通用指针

下面给出通用指针定义说明格式的主要部分:

类型说明符 \* 标识符;

其中:

(1) 指针定义说明未加空间修饰符。凡指针定义说明未加空间修饰符的编译器一律预留三个字节作为通用指针对待。

(2) 通用指针在编辑和连接/定位时,才把存储空间码和地址填入预留的三字节中。三字节的分配如下:从低地址单元开始,先放存储空间代码(idata 为 1, xdata 为 2, pdata 为 3, data 为 4, code 为 5);其次,放 16 位地址,先高字节,后低字节。如果是 8 位地址,高字节补 0。参见图 7.1。

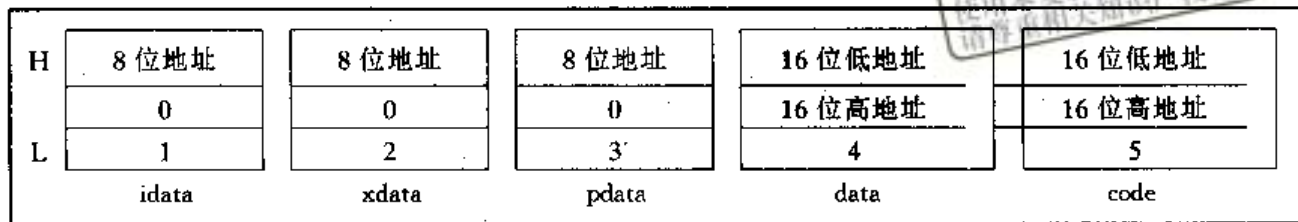


图 7.1 通用指针定位到具体存储空间

[例 7.9]

```
char *pc; /* pc 作为通用指针预留三字节 */
```

编译时若使用大模式,对象被定位在 xdata 空间。通用指针的三字节为:

高地址	指针低字节
	指针高字节
低地址	2

[例 7.10]

```
extern int printf(void *format, ...);
int xdata *px;
char code *fmt = "value = %d, 0x%x \n";
printf(fmt, *px, *px);
```

本例中 printf(void \*format, ...) 的参数表中的指针未定位存储空间,作为通用指针预留了三字节。在 printf() 被编辑和连接/定位时才把存储空间代码和地址填入三字节,指针内容为:5、指针高字节、指针低字节。

### 7.3.2 抽象指针——匿名指针

抽象指针是:把绝对地址用带有存储空间修饰符的指针类型加以强制,当作已赋值的指针用于对象的存取(叫做绝对存取)和函数调用(叫做绝对调用)。由于是把直接地址当作赋值指针来用但又并未起名,所以叫做匿名指针。

[例 7.11]

```
int i;
i = *((int)xdata *)0x8000;
```

[例 7.12] 在 code 空间 0xFF00 处有一段子程序代码。该子程序代码无名无参但返回 int 量,试用匿名指针进行调用。

```
int i;
i = (in (code *) (void))0xFF00;
```

### 7.3.3 指针可用运算符

指针可用的运算符有限,包括:

加 +	限于指针 + int 量
减 -	限于指针 - int 量
++	指针 ++, ++ 指针
--	指针 --, -- 指针
赋值 =	允许将 unsigned long 常量赋预指针
关系运算符	=, !=, <, <=, >, >=





## 第八章 C51 存储模式

### 8.1 C51 三种存储模式



为适应不同规模的程序,在编译 C51 源程序时可选用三种存储模式之一,即小模式 (small)、紧凑模式(compact)或大模式(large)。

存储模式决定所用程序存储空间和数据存储空间的大小,详见表 8.1。

表 8.1 存储模式与缺省存储空间

存储空间 存储模式 空间大小	程序存储空间	数据存储空间					寄存器
	code 片内外 混合编址	data	idata	bdata	pdata	xdata	堆栈 组 0~3 或按绝对地址
small	64 KB	128 B (0~256 B)	16(128 位)				idata 4 * 8
compact	64 KB	128 B (0~256 B)	16(128 位)	256 B			pdata 4 * 8
large	64 KB	128 B (0~256 B)	16(128 位)			64 KB	xdata 4 * 8
面向指针的存储空间码	5	4	1		3	2	

讨论:

(1) C51 中三种模式的 ROM 空间相同,都是 64 KB,只有 RAM 空间不同。所有 8051 系列芯片片上 RAM 至少有 data(128 B) 空间;至于片上 RAM 的 idata(0~256 B) 空间则因芯片而异。片上 RAM 空间只与芯片有关,不会随存储模式而变。

(2) 三种模式的缺省 RAM 空间:

- 对于 small 模式就是片上 RAM 的所用空间(即 data(128 B)和 idata(0~256 B))。
- 对于 compact 模式为片外 pdata(256 B) 空间,有的芯片已做到片内。
- 对于 large 模式为片外 xdata(64KB) 空间。

(3) 表 8.1 用下划线标出三种模式的缺省 RAM 空间。

(4) 三种模式的堆栈空间也用下划线标出。

(5) 在程序中,各种数据在定义时只要未用修饰符指明存储空间的,都分配在 RAM 缺省空间。

(6) 对于函数的参数和函数内部定义的自动变量,因它们是可被覆盖重用的 (overlayable);故放在堆栈之中。

(7) 为重入函数而设的重入栈也设在各缺省存储空间中。

(8) data 空间的 0~1FH,是四组通用寄存器组。它们可按 RB=0~3 分组,每组有 R0~R7;也可留 0 组按 R0~R7 使用,其余按绝对地址使用。

(9) 未被存储模式选为缺省的存储空间,只要物理上存在,都可以由程序员安排全局变量或静态变量使用,只要在定义说明时加上存储空间予以修饰即可。

(10) 三种存储模式中有的访问效率和代码长度上出现不利,为此,允许使用混合模式。即把经常访问的变量不管存储模式而强制放到片内;大块数据放在片外而把它的指针放在片内;为指针指定对象所在的存储空间,既有利加快存取速度,又不至采用三字节的指针。

## 8.2 C51 内部对数据和函数的组织规范

### 8.2.1 标识符改大写字符和函数换名

C51 内部对标识符的管理规范:

• C51 对所有定义说明的数据的标识符转换为大写字符进行存放。

• C51 对所有定义说明的函数除去将标识符转换为大写字符外,还对函数有无寄存器参数传送和函数是否可重入进行换名。函数换名的规则如下:

func( void )	FUNC	无参函数名只改为大写
func(参数表)	FUNC	无寄存器传送参数的函数名只改为大写
func(参数表)	_FUNC	有寄存器传送参数的改为大写并加“_”前缀
func(void) reentrant	? FUNC	无参数和可重入的改为大写并加“?”前缀
func(参数表)reentrant	_? FUNC	有寄存器传送参数和可重入的改为大写并加“_?”前缀

前缀

### 8.2.2 全局变量存放的段名规定

下面按多文件程序考虑。每个文件为一个编译模块。编译模块以文件主名命名(即不带扩展名),称为编译模块名。

全局变量按常量及所在存储空间分类而放入与其编译模块有关的各段(seg)中,见表 8.2。

表 8.2 编译模块的全局变量段名

段 名	段中所放全局变量
? DT? 模块名	本模块全局 DATA 变量
? ID? 模块名	本模块全局 IDATA 变量
? PD? 模块名	本模块全局 PDATA 变量
? XD? 模块名	本模块全局 XDATA 变量
? BI? 模块名	本模块全局 BIT 变量
? BA? 模块名	本模块全局按字节位寻址变量
? CC? 模块名	常量 C 字符串和带初值变量

### 8.2.3 函数的段名

函数有程序部分和局部数据部分。它们分别放在独立的段中。表 8.3 给出各存储模式下

每个函数使用的段名。

对函数的代码段和数据段(包括位数据),只要未加 `reentrance` 修饰符,编译时都加上了可覆盖属性(overlayable)。标注覆盖属性的函数空间,在连接时是被覆盖而重复使用的。对于覆盖属性的 DATA 段和覆盖属性的 BIT 段的首地址已被预定义为“? 函数名? BYTE 和 ? 函数名? BIT”,并为公用性质。

表 8.3 各存储模式下每个函数使用的独立段名

存储模式	段 名	段空间名	用 途
SMALL	? PR? 函数名? 模块名	CODE 段	放函数代码
	? DT? 函数名? 模块名	DATA 段	放局部变量
	? BI? 函数名? 模块名	BIT 段	放局部位变量
COMPCT	PR? 函数名? 模块名	CODE 段	放函数代码
	? PD? 函数名? 模块名	PDATA 段	放局部变量
	? BI? 函数名? 模块名	BIT 段	放局部位变量
LARGE	? PR? 函数名? 模块名	CODE 段	放函数代码
	? XD? 函数名? 模块名	XDTA 段	放局部变量
	? BI? 函数名? 模块名	BIT	放局部位变量

注:各种模式数据段和位段的首地址预定义公用名: ? 函数名? BYTE 和 ? 函数名? BIT

#### 8.2.4 函数的参数传送规则

函数参数的传送,在 C51 中,规定前三个参数要尽可能地按表 8.4 的规则选用寄存器传送。寄存器不够用或参数多于三个时放在各模式对应的缺省数据段中。

表 8.4 函数前三个参数使用寄存器传送的规则

参 数 \ 类 型 REG	char 或 一字节指针	int 或 二字节指针	long float	通用指针(三字节)
第一参数	R7	R6, R7 (H1) (L0)	(MMM E/S) R4~R7 (MSB) (LSB)	R1, R2, R3 (L0) (H1) (码)
第二参数	R5	R4, R5 (H1) (L0)	R4~R7	R1, R2, R3
第三参数	R3	R2, R3 (H1) (L0)	无	R1, R2, R3

[例 8.1] `fundcl(int a, intb, int *c) { ... }`

第一参数 a 用 R6~R7 传送;

第二参数 b 用 R4~R5 传送;

第三参数 c 用 R1~R3 传送。

[例 8.2] func1( long d, long e) {...

第一参数 d 用 R4~R7 传送;

第二参数 e 用模式缺省数据段传送。

[例 8.3] func3( float f, char g) {...

第一参数 f 用 R4~R7 传送;

第二参数 g 用传送模式缺省数据段。

### 8.2.5 重入栈的有关规定

C51 有关重入栈的规定为:

- 可重入函数需要专门的重入栈。重入栈所在存储空间与存储模式有关。
- 表 8.5 列出重入栈所在空间和重入栈的栈指针。
- 重入栈的栈指针是启动文件(STARUP.A51)中的预定义全局变量。
- 重入栈与一般函数栈不同,是倒过来自顶向下堆放的。

表 8.5 各存储模式重入栈有关规定

存储模式	重入栈区位置	重入栈指针预定义名
SMALL	idata(256 B)	? C_IBP(一字节长)
COMPACT	pdata(256 B)	? C_PBP(一字节长)
LARGE	xdata(64 KB)	? C_XBP(二字节长)

注:重入栈自顶向下堆放。

### 8.2.6 函数返回值的規定

在 C51 中,函数返回值一律用寄存器返回。返回的规则见表 8.6。

表 8.6 函数返回值指定用寄存器

返回值类型	寄存器	注
bit	CY	
unsignedchar	R7	
unsignedint	R6, R7	R6 放高位, R7 放低位
unsignedlong	R4~R7	R4 放最高位, R7 放最低位
float	R4~R7	IEEE 标准 R7 放符号位及阶码
指针	R1, R2, R3	R3 放存储空间码 R2 放高位 R1 放低位

## 第九章 C51 函数及库函数

### 9.1 函数说明

C51 函数的说明如下：

#### 1. 定义性说明

[存储类说明符⑦] 类型说明符① 标识符② (形参表①) [reentrant③] [interrupt m④]  
[using n⑦] [存储模式⑤] | ... ① |

⑦	①	②	⑤
{static extern}	{unsigned char char unsigned int int unsigned long long float struct union void bit}	{函数名 * 函数名 ( * 函数名) * ( * 函数名)}	{small compact large}

#### 2. 引用性原型说明格式

extern [存储类说明符] 类型说明符 标识符 (形参表) [reentrant] [using n] [存储模式]

#### 3. 函数调用格式

标识符 (实参表);

或 x = 标识符 (实参表);

其中：

(1) 类型说明①部分——与基础部分的返回类型说明符相比，浮点数只有 32 位的 float，没有 64 位的 double，但多了位类型 bit。

(2) 标识符②部分——由标识符与指针符 \* 组合而成：

函数名 ( )	带或不带返回值的函数
* 函数名 ( )	返回指针的函数
( * 函数名) (void)	定义带或不带返回值的无参函数指针
( * 函数名) (形参表)	定义带或不带返回值的有参函数指针
* ( * 函数名) (void)	定义返回指针的无参函数指针
* ( * 函数名) (形参表)	定义返回指针的有参函数指针

(3) 存储类说明符①部分——定义性说明可加 static 使函数对本文件外隐蔽和对本文件本定义之前的部分隐蔽。C 中所有定义的函数都具有 extern 性质(即外部可用), 所以 extern 定义时是被省略的。但在引用性说明中 extern 是必须的。

(4) 实参表①部分——最好保持与基础部分一样, 即参数中不要加存储空间修饰符, 以便参数所在空间能随存储模式而自动进入缺省空间。惟独对于指针, 有时为了不用最长的通用指针, 可为指针指向的对象指定空间。但应保证该指针所指向对象的空间是正确的。如:

```
void fxp( char xdata * xp, int idx ) { ... }
```

(5) 修饰符②③④⑤部分——关于 C51 函数的修饰符请见后续各节。虽然它们同属修饰符, 但是, 由于它们同时可以选用多个, 所以, 在格式中把它们罗列起来。

(6) 函数体⑥部分无须多加解释。

## 9.2 函数被修饰使用指定的寄存器组

8051 有 4 个寄存器组, 每组有寄存器 R0~R7。C51 中, 为函数作定义性说明时可以指定函数内部使用的寄存器组。其定义性说明如下:

```
void 标识符(实参表) using n { ... }
```

其中:

(1)  $n=0\sim3$  为寄存器组号。

(2) 加入修饰部分 using  $n$  之后, C51 自动在函数的汇编码中加入如下的函数头段和函数尾段:

```

pushpsw
movpsw, #与寄存器组号 n 有关的常量 ;(psw & ~0x18) & n * 8
...
pop psw

```

(2) using  $n$  不能用于有返回值的函数。因为, C51 的返回值是放在寄存器中的, 而返回前寄存器组却被改变了, 致返回值不对。

(3) 有两个影响寄存器组的编译伪指令:

```
#pragma REGISTERBANK(n) 选用寄存器组 n
```

```
#pragma AREGS/NOAREGS 选用/停用寄存器绝对地址
```

它们原则上在程序的任何地方都是可用的, 但惟独不要用于函数之中。因为它们会破坏函数内部寄存器的完整性。

(4) 如果函数要使用另外一组寄存器, 而函数又必须有返回值, 这时, 不要给函数加修饰符 using  $n$ , 而要改用如下的编译伪指令并放在调用函数之前和之后:

```
#pragma REGISTERBANK(n)
```

调用函数之后所加的伪指令应放在返回值已被取出之后, 且这次伪指令中的  $n$  应与调用函数之前的寄存器组不同。

(5) 引用性原型说明中不要加寄存器修饰符, 避免误写  $n$  值, 导致混乱。

(6) 函数被 using  $n$  修饰, 但函数中又未用到通用寄存器, 优化时, 程序头段中关于修改



寄存器组的部分会被优化掉。



### 9.3 函数被修饰为中断函数

C51 定义函数时加入 `interrupt m` 修饰符可将函数转化为中断函数。这时候, C51 自动加入汇编码中断程序头段和中断程序尾段, 并填写好中断向量表中对应的表项。

中断函数定义性说明格式如下:

```
void 标识符(void) interrupt m {...}
```

其中:

(1)  $m = 0 \sim 31$ , 0 对应于外部中断 0;

1 对应于片内 `TIMER0` 中断;

2 对应于外部中断 1;

3 对应于片内 `TIMER1` 中断;

4 对应于片内串行口中断。

其他为预留。

(2) 中断的函数必须是无参数无返回量的函数。

(3) 中断函数体中可以调用其他函数, 但应保持其寄存器组与中断函数的一致。在中断函数和被调函数都未使用 `using n` 修饰符的情况下, C51 负责选择一组合适的寄存器组作为绝对寄存器供它们共同访问, 保持其一致性。如果使用了 `using n` 修饰符, 则保证一致性转由程序员负责, C51 不作检查与报警。

(4) 函数体中可以用浮点数运算。遇需要保存和恢复浮点寄存器状态时, 函数库中提供了 `fp_save()` 和 `fp_restore()` 可以使用(见 `math.h` 文件)。

(5) C51 为中断函数自动添加汇编语言的程序头段和尾段。内容有:

·头段中保护 `ACC, B, DPH, DPL` 和 `PSW` 入栈, 尾段中恢复出栈。

·中断函数未同时加 `using n` 修饰符的, 要将 `R0 ~ R7` 在头段中入栈, 在尾段中出栈。加 `using n` 修饰符的, 在头段中将 `PSW` 入栈之后要修改 `PSW` 中的寄存器组号部分。

·将函数体的最后一条汇编指令 `RET` 修改为 `RETI`。

(6) C51 会自动把中断函数入口地址添入中断向量表的对应于该中断的表项中。表项的偏移地址为程序存储空间的:  $m * 8 + 3$ 。

(7) 可以用 `#pragma NOINTVECTOR` 解除中断向量表的自动填入, 以增加编程的灵活性。但这时向量表的填入由程序员负责。譬如, 编写某一中断的接管程序时, 需要先把原中断向量的表项内容保存起来, 才可填入新的入口地址。这时候向量表的自动填入, 需要解除。

(8) 中断函数已把 `RET` 指令自动修改为中断返回指令 `RETI`(它将影响硬件中断系统的状态), 所以, 应禁止任何形式对中断函数的直接调用, 以避免硬件中断系统的混乱。C51 能识别对中断函数的直接调用, 并予以拒绝。但是, 用指针的间接调用难以查觉, 应当避免。

(9) 中断函数最好写在文件的尾部, 并且禁止使用 `extern` 存储类说明符, 杜绝软件调用的可能性。

[例 9.1] 下面是用 C51 编写的中断函数示例。

```
extern bit alarm;
```



```
int alarm_count;
```

```
void falarm() interrupt 1 using 1
{
    alarm_count *= 2;
    alarm = 1;
}
```

经 C51 编译后,产生如下的汇编代码:

```
                ;FUNCTION FALAARM (BEGIN)
0000 C0E0      PUSH      ACC
0002 C0D0      PUSH      PSW
0004 E500  R  MOV       A, alarm_count+01H
0006 25E0      ADD       A, ACC
0008 F500  R  MOV       alarm_count+01H, A
000A E500  R  MOV       A, alarm_count
000C 33       RLC       A
000D F500  R  MOV       alarm_count, A
000F D200  E  SETB      alarm
0011 D0D0      POP       PSW
0013 D0E0      POP       ACC
0015 32       RETI
                ;FUNCTION FALAARM (END)
```

本例虽然使用了 using 1,但是,实际上并未使用工作寄存器,因此,切换寄存器组的代码被优化掉了。

## 9.4 函数被修饰为重入函数

允许被嵌套调用的函数必须是可重入函数。函数的嵌套调用是指当一个函数正被调用尚未返回,又被本函数或其他函数再次调用,只有等到后次调用返回到了本次,本次被暂时搁置的程序才得以正确地恢复接续原来的正常运行,直到本次返回。一般的函数做不到这一点,因为它不是可重入的函数。重入函数必须具有分层保护参数和局部变量的专门重入堆栈机制,才可被嵌套调用。多任务系统的不同任务之间经常发生对一个函数的嵌套调用。对于单任务系统,本函数被直接或间接地自我递归调用也是函数的嵌套调用。只有重入函数才允许嵌套调用。重入函数的定义性说明如下:

类型说明符 标识符(实参表) reentrant {...}

其中:

(1) reentrant 重入修饰符是不可缺少的。

(2) 实参表中不允许使用 bit 类型的参数。函数体中也不允许存在含有任何关于位变量的操作。当然,更不能返回 bit 类型。

(3) 重入函数不使用一般函数位于存储模式缺省空间的覆盖式堆栈,而是在此同一缺省空间内从顶端另行分配一个非覆盖的重入堆栈。它与覆盖式堆栈相向,但自上而下地堆放。重入堆栈将嵌套调用的每层参数及局部变量一直保留到控制由深层返回本层,而又终止于本

层的返回。各模式的重入栈空间和其堆栈指针见表 8.5。

(4) 一个函数被嵌套调用时,重入栈的空间消耗很大,对于 8051 微控制器应有节制使用。

(5) 重入函数可以与存储模式修饰符同时使用。模块中同时存在不同存储模式的重入函数时,将占用不同空间的不同重入栈。

(6) 重入函数的递归调用链中可以存在中断函数对它的调用。

[例 9.2] 定义说明:

```
int calc(char i,int b) reentrant
{
    extern int table[ ];
    int x;
    x = table[i];
    return (x * b);
}
```

调用:

```
extern int table[ ];
char i,j;
int factor,result;
i=2;j=1;factor=10;
result=calc(i,calc(j,factor));
```

## 9.5 函数被修饰为使用指定的存储模式

前边已经讲过,为了提高程序的效率和灵活性,允许采用混合存储模式。为函数指定了固定的存储模式,它将不再随编译模式而变。

为函数指定存储模式的格式如下:

类型说明符 标识符(形参表) 存储模式修饰符 {...}

```
[small
compact
large]
```

其中:

- (1) 存储模式修饰符必不可少。
- (2) 它可用 small, compact, large 中的一个。
- (3) 存储模式为本函数的参数和局部变量指定不随编译模式而变的固定存储空间。
- (4) 如果本函数同时修饰为重入函数,则重入栈也在这个存储空间中。

[例 9.3]

```
#pragmalarge /* 按大模式编译 */
extern intcalc(char i,int b) small reentrant; /* 修饰为小模式 */
extern int func(int i,float f) large; /* 修饰为大模式 */
extern void *tcp(char xdata *xp,int ndx) small; /* 修饰为小模式 */
```

```

int mtest(int i, int y) small          /* 修饰为小模式 */
{
    return(calc(2), calc(6, y) + func(-1, 4.75));
}

int func1(int i, int y)               /* 因编译取得大模式 */
{
    return ( mtest( i, k ) + 2);
}

```



## 9.6 C51 与 PL/M51 函数的交叉调用

C51 支持与 PL/M51 在函数方面的兼容。因为同属最贴近硬件的高级语言,互相调用比较简单。C51 调用 PL/M51 写的函数只要按 C51 的格式写引用性说明,但需在 `extern` 之后加上 `alien` (异形)关键字。如果是 PL/M51 调用 C51 写的函数,只要在 C51 函数的定义性说明的最前面加上 `alien` (异形)关键字即可。

格式如下:

```

extern alien char plm_func(unsigned char, unsigned int);
alien char c_func(unsigned char x, unsigned char y) {;}

```

其中:

(1) 第一种格式是引用异形(`alien`)函数的原型说明。它表明要调用的函数是用 PL/M51 写的。但是,为要能把 PL/M51 的函数写成 C51 的格式,需要对 PL/M51 有所了解。

(2) 第二种格式是 C51 中供 PL/M51 调用的异形(`alien`)函数的定义性说明。C51 程序员书写时不会有什么困难,因为只在说明的最前面加 `alien` 而已,剩下来的事由编译器解决。

(3) C51 中已修饰为 `reentrant` 的函数不应再加上 `alien` 说明为被 PL/M51 调用的函数。即重入函数不应被 PL/M51 调用,因为 PL/M51 不支持重入。

## 9.7 C 与汇编函数的交叉调用

为了发挥 C 与汇编两种语言各自的优势,希望能够实现它们的混合编程。两种语言接口的关键是不同语言函数的交叉调用。只要两种语言用同一规范书写函数,用同一规范调用函数就可以了。C 语言对函数的参数和返回值的传送规则,以及对代码段、数据段、位段的选用和命名都有严格的规定。所以,汇编语言要向 C 语言关于函数和调用的规范看齐。即在汇编语言方面,无论写汇编子程序和调 C 的函数都得遵守 C 的规范,只不过换成汇编语言来描述而已。关于 C 语言定义函数和调用函数的内部安排已在第三章做了详细介绍。这里再择其要归纳一下。

### 1. 编写被 C 调用的汇编子程序

(1) 要获得由 C 输入的函数参数,前三个参数一般应从寄存器中取得。寄存器不够或参数多于三个时,则是放在与模式有关的缺省数据段中传入的。数据段的首地址作为公用变量已经命名,可以使用。数据是按定义说明的先后顺序安放的。对于位变量另有位段。

(2) 汇编子程序的主体部分放在命名的 `code` 段中。主体部分的一开始一定要把寄存器中

的参数保存起来,因为函数内部可能要使用这些寄存器。函数的内部变量应安排在与参数相同的段内。

(3) 函数的主体部分用汇编写出。

(4) 函数的结尾部应考虑是否有返回量。如有,在 RET 指令之前放入合适的寄存器中;没有返回量,则直接写 RET 指令返回。

(5) 应将函数地址和数据段及位段的首址均说明为汇编的 public。

(6) 将程序段、数据段和位段均应加 overlayable 的连接属性。

## 2. 用汇编编写调用 C 的函数

(1) 向 C 的函数传入参数:前三个参数放寄存器,寄存器放不下的和多于三个的参数放在与函数有关的数据段中。参数是从公用的数据段首地址开始顺序安放的。

(2) 用 LCALL 调函数地址(即 C 语言的函数名)。请注意存在经寄存器传送的参数时,函数名前应加下划线。reentrant 函数加 ? 号。

(3) 如 C 函数有返回值,在汇编的 LCALL 之后,应根据情况是否从寄存器取出返回值。

(4) 在程序的开始部分应把所欲调的函数名(注意是否加下划线和 ? 号前缀)和数据段的首址说明成 EXTRN。

下面写出调 C 函数的汇编码和供 C 调用的汇编码子程序。因为存储模式影响变量的存储空间,从而也影响到存取变量的指令不同,如:

small 模式     参数和局部变量放 data 空间,用 MOVE 存取。

compact 模式     参数和局部变量放 pdata 空间,用 MOVX @RI 存取。

large 模式     参数和局部变量放 xdata 空间,用 MOVX @DPTR 存取。

所以,下例中分存储模式给出代码。

[例 9.4] 设有函数原型(C 语言格式)如下:

```
int function(int v_a, char v_b, bit v_c, long v_d, bit v_e);
```

试写出 C 与汇编两种语言的交叉调用。

### 1. small 模式

#### (1) 汇编调 C 函数:

```
EXTRN (_function)           ; 外部函数
EXTRN DATA (? -function? BYTE) ; 局部变量首地址(见表 3.3 注)
EXTRN BIT (? -function? BIT)   ; 局部位变量首地址(见表 3.3 注)
```

```
...
MOV     R6, # HIGH v_a
MOV     R7, # LOW v_a
MOV     R5, # v_b
MOV     C, v_c
MOV     ? -function? BIT + 1, C
MOV     ? -function? BYTE + 3, v_d + 0
MOV     ? -function? BYTE + 4, v_d + 1
MOV     ? -function? BYTE + 5, v_d + 2
MOV     ? -function? BYTE + 6, v_d + 3
MOV     C, v_e
```



```

MOV      ? - function? BIT + 1, C
LCALL    - function
MOV      intresult + 0, R6      ; 接受返回 int 值
MOV      intresult + 1, R7
...

```



## (2) 供 C 调用的汇编子程序:

```

? PR? FUNCTION? MODULE SEGMENT CODE
? DT? FUNCTION? MODULE SEGMENT DATA OVERLAYABLE
? BI? FUNCTION? MODULE SEGMENT BITO VERLAYABLE

```

```
PUBLIC _function, (- ? function? BYTE), (- ? function? BIT)
```

```

RSEG ? DT? FUNCTION? MODULE
(? - function? BYTE):

```

```

v_a: DS 2
v_b: DS 1
v_d: DS 4

```

```

RSEG ? BI? FUNCTION? MODULE
(? - function? BIT):

```

```

v_c: DBIT1
v_e: DBIT1

```

```
RSEG ? PR? FUNCTION? MODULE
```

```

- function:      MOV      v_a, R6
                  MOV      v_a + 1, R7
                  MOV      v_b, R5
                  ...
                  ; 汇编子程序主体部分
                  MOV      R6, # HIGH reurnval
                  MOV      R7, # LOW returnval
                  RET

```

## 2. compact 模式

### (1) 汇编调 C 函数:

```

EXTRN (_function)      ; 外部函数
EXTRN XDATA (? - function? BYTE) ; 局部变量首地址
EXTRN BIT (? - function ? BIT) ; 局部位变量首地址
...
MOV      R6, # HIGH v_a
MOV      R7, # LOW v_a
MOV      R5, # v_b
MOV      C, v_c
MOV      ? - function? BIT + 1, C
MOV      R0, # ? - function? BYTE + 3
MOV      A, v_d

```

```

MOVX    @R0, A
INC     R0
MOV     A, v_d + 1
MOV     X @R0, A
INC     R0
MOV     A, v_d + 2
MOVX    @R0, A
INC     R0
MOV     A, v_d + 3
MOVX    @R0, A
MOV     C, v_e
MOV     ? - function? BIT + 1, C
LCALL   - function
MOV     intresult + 0, R6      ; 接受返回 int 值
MOV     intresult + 1, R7
...

```



(2) 供 C 调用的汇编子程序:

```

? PR? FUNCTION? MODULE SEGMENT CODE
? PD? FUNCTION? MODULE SEGMENT XDATA OVERLAYABLE IPAGE
? BI? FUNCTION? MODULE SEGMENT BIT OVERLAYABLE

```

```

PUBLIC _function, (-? function? BYTE), (-? function? BIT)

```

```

RSEG ? PD? FUNCTION? MODULE
(? - function? BYTE):

```

```

    v_a DS 2
    v_b DS 1
    v_d DS 4

```

```

RSEG ? BI? FUNCTION? MODULE
(? - function? BIT):

```

```

    v_c DBIT1
    v_e DBIT1

```

```

RSEG ? PR? FUNCTION? MODULE

```

```

-function: MOV R0, # ? - function? BYTE

```

```

    MOV     A, R6
    MOVX    @R0, A
    INC     R0
    MOV     A, R7
    MOV     X@R0, A
    MOV     A, R5
    INC     R0
    MOVX    @R0, A

```

...

; 汇编子程序主体部分



```
MOV    R6, #HIGH reurnval
MOV    R7, #LOW returnval
RET
```

### 3. large 模式

#### (1) 汇编调C函数:

```
EXTRN  (_function)                ; 外部函数
EXTRNX DATA (? -function? BYTE) ; 局部变量首地址
EXTRN  BIT (? -function ? BIT)    ; 局部位变量首地址
...
MOV    R6, #HIGH v_a
MOV    R7, #LOW v_a
MOV    R5, # v_b
MOV    C, v_c
MOV    ? -function? BIT + 1, C
MOV    DPTR, #? -function? BYTE + 3
MOV    A, v_d
MOVX   @DPTR, A
INC    DPTR
MOV    A, v_d + 1
MOVX   @DPTR, A
INC    DPTR
MOV    A, v_d + 2
MOVX   @DPTR, A
INC    DPTR
MOV    A, v_d + 3
MOVX   @DPTR, A
MOV    C, v_e
MOV    ? -function? BIT + 1, C
LCALL  -function
MOV    intresult + 0, R6;          接受返回 int 值
MOV    intresult + 1, R7
...
```

#### (2) 供 C 调用的汇编子程序:

```
? PR? FUNCTION? MODULE SEGMENT CODE
? PD? FUNCTION? MODULE SEGMENT XDATA OVERLAYABLE
? BI? FUNCTION? MODULE SEGMENT BIT OVERLAYABLE
```

```
PUBLIC _function, (? -function? BYTE), (? -function? BIT)
```

```
RSEG ? XD? FUNCTION? MODULE
(? -function? BYTE);
```





```

v_a DS 2
v_b DS 1
v_d DS 4
RSEG ? BI? FUNCTION? MODULE
(? - function? BIT):
v_c DBIT1
v_e DBIT1
RSEG ? PR? FUNCTION? MODULE
- function:    MOV     DPPTR, # ? - function? BYTE
               MOV     A, R6
               MOVX    @DPPTR, A
               INC     DPPTR
               MOV     A, R7
               MOV     X@DPPTR, A
               MOV     A, R5
               INC     DPPTR
               MOV     X@DPPTR, A
               ...
               ; 汇编子程序主体部分
               MOV     R6, # HIGH reurnval
               MOV     R7, # LOW returnval
               RET

```

## 9.8 内部函数

有这样一些函数,用汇编语言编写,非常直接、简单而且目标码很短;而用 C 语言编写,却目标码很长。对于用汇编语言编写的这类库函数叫做内部函数。内部函数,在 C51 中,已经按 C 的规范用汇编语言写好。用户可以直接当作 C 语言函数调用它们。内部函数已经放在库中,供程序员使用。内部函数的原型说明放在 `intrans.h` 头文件中。

C51 的内部函数有:

<code>_crol _</code>	<code>_cror _</code>	无符号字符型变量左/右移位
<code>_irol _</code>	<code>_iror _</code>	无符号整型变量左/右移位
<code>_lrol _</code>	<code>_lror _</code>	无符号长整型变量左/右移位
<code>_nop _</code>		空操作
<code>_testbit _</code>		位测试

内部函数在书写上有共同特点,就是在函数名的前后都加有下划线,以与其他 C 的函数相区别。

### 9.8.1 左移多位函数

下面给这类内部函数的原型说明:

```

unsigned char _crol_( unsigned char val, unsigned char n );
unsigned int  _irol_( unsigned val, unsigned char n );

```

```
unsigned long _lrol_( unsigned long val, unsigned char n )
```

其中:

函数的第一个参数是被移位的变量。第二参数是欲移位的位数,对于无符号字符型变量为 0~7,无符号整型变量为 0~15,无符号长整型变量为 0~31。

[例 9.5]

```
#include <intrins.h>

void main( )
{ unsigned int y;
  y=0x00FF;
  y=_lrol_(y,4); /* y=0x0FF0 */
}
```

### 9.8.2 右移多位函数

下面给出这类内部函数的原型说明:

```
unsigned char _rcor_( unsigned char val, unsigned char n );
unsigned int _iror_( unsigned val, unsigned char n );
unsigned long _lror_( unsigned long val, unsigned char n );
```

其中:

函数的第一参数是被移位的变量。第二参数是欲移位的位数。对于无符号字符变量为 0~7,无符号整型变量为 0~32,无符号长整型变量为 0~31。

[例 9.6]

```
#include <intrins.h>

void main( )
{ unsigned int y;
  y=0x00FF;
  y=_iror_(y,4); /* y=0x000F */
}
```

### 9.8.3 空操作函数

函数原型说明格式:

```
void _nop_(void);
```

本函数产生单一汇编指令 `nop`。执行它实际上没有实质性的操作,仅只是延时一个机器周期。

[例 9.7] 从 P0.7 输出三个机器周期宽的正脉冲。

```
p0 &= ~0x80;
p0 |= 0x80;
_nop_;
_nop_;
p0 &= ~0x80;
```

### 9.8.4 位测试函数

函数原型说明格式：

```
bit _testbit_(bit x);
```

其中：

参数和返回值必须是位变量。

本函数产生汇编指令 JBC x, --。用于测试位变量是 x, 是 0, 还是 1, 并将其值经 CY 返回。

[例 9.8]

```
1  #include <intrins.h>
2
3  bit flag;
4  char val;
5
6  void main( )
7  { if (! _testbit_(flag))
8    val--;
9  }
```

编译后：

第 7 行是： JBCflag, ? C002

第 8 行是： DECval

第 9 行是：? C002: RET

## 9.9 抽象数组(绝对地址存取)——absacc 库函数

本节介绍利用三字节通用指针作为抽象指针, 为各存储空间提供绝对地址存取的技术。方法是把通用指针指向各存储空间的首址, 并按存取对象类型进行指针强制, 再用定义宏说明为数组名即可。存取时利用数组的下标变量寻址。

具体实现：

用预处理器伪指令 #define 为各空间的绝对地址定义宏数组名：

```
#define CBYTE ((unsigned char *)0x50000L) /* code 空间 */
#define DBYTE ((unsigned char *)0x40000L) /* data 空间 */
#define PBYTE ((unsigned char *)0x30000L) /* pdata 空间 */
#define XBYTE ((unsigned char *)0x20000L) /* xdata 空间 */
```

以上存取对象是 char 类型字节。下面存取对象是 int 类型, 8051 中叫字。

```
#define CWORD ((unsigned int *)0x50000L) /* code 空间 */
#define DWORD ((unsigned int *)0x40000L) /* data 空间 */
#define PWORD ((unsigned int *)0x30000L) /* pdata 空间 */
#define XWORD ((unsigned int *)0x20000L) /* xdata 空间 */
```

绝对地址对象的存取, 用指定下标的抽象数组元素来实现如下：



char 类型:	CBYTE[i]	int 类型:	CWORD[i]
	DBYTE[i]		DWORD[i]
	PBYTE[i]		PWORD[i]
	XBYTE[i]		XWORD[i]



其中:

因为定义的宏数组名为各空间的绝对零地址,所以带下标变量  $i$  的数组元素就是各空间绝对地址的内容。

[例 9.9] DBYTE [0x10] 为 data 空间绝对地址 16 处的字节对象。

XWORD [0xFF] 为 xdata 空间绝对地址 255 处的字对象。

上述抽象数组名的宏定义已在头文件 absacc.h 中说明。

抽象数组元素与前面介绍过的抽象指针有如下的关系:

$XWORD [0xFF] = *(int \ xdata * )0xFF ;$

## 9.10 C51 库函数介绍

C51 函数库与 ANSI C 的库函数的函数名、功能及参数尽量做到一致。不同之处有:

1. 增加了 ANSI C 没有的内部函数和抽象数组。它们已在前面介绍过了,其头文件分别是 intrans.h 和 absacc.h。

2. 参数和返回值的类型可能与 ANSI C 有出入。这是为考虑到 8051 的硬件特点和指令特点,以求得库函数的时、空高效。为此,凡小数据类型能满足的不用大数据类型,能用 bit 的不用 char, int...8051 没有符号数运算指令,凡涉及数学运算的变量能用 unsigned 的不用 signed。关于函数的返回值多用 bit 类型。

3. 受 8051 各存储空间容量有限的制约,对具有可变参数的库函数,做了最大参数数量的限制,如对 small 和 compact 模式为 15 个字节参数量(即参数全部为指针时,最多 5 个;长整型参数最多能传 3 个,余下字符用于其他);对 large 模型为 40 个字符。

4. 所有函数独立于寄存器组。即库函数均未指定寄存器组,故能适应于任一寄存器组。

5. ANSI C 中标准的 I/O 设备是以控制台(console)为标准输入和输出设备的。8051 是以串行口为字符的输入输出设备的。所以,并不直接支持系统机中所谓的控制台。为使 getkey 和 putchar 等函数能够工作,使用之前一定要对 8051 的串行口做初始化工作。如对于 12MHz 的 8051,在波特率为 2400 时,应做如下的初始化:

SCON = 0x52;

TMOD = 0x20;

TRI = 1;

THI = 0xF3;

6. 如果使用矩阵式小键盘作输入,使用 LED、LCD 或微型打印机做输出,应自行重新编写适合它们的驱动程序并改写 getchar 和 putchar。

7. 8051 未提供有关文件管理,磁盘管理等库函数。因为对于 8051 尚无此需要。

8. 函数中,凡宏替换比函数调用效率更好的,库函数的所给经常是宏。宏定义基本上是放在相应的头文件中。

下面按头文件分类列出 3.20 版以前的全部库函数名,并给出其主要属性。见表 9.1。

库函数的功能、原型和示例请见附录 A。库函数的示例需要不断的查阅和熟悉,请读者务必注意,不要忽视。

表 9.1 C51 库函数汇总表

函数名	头文件	属性
<b>1. 数学函数</b>		
abs	math.h	reentrant
cabs	math.h	reentrant
fab	math.h	reentrant
labs	math.h	reentrant
exp	math.h	non_reentrant
log	math.h	non_reentrant
log10	math.h	non_reentrant
sqr	math.h	non_reentrant
rand	math.h	reentrant
srand	math.h	non_reentrant
cos	math.h	non_reentrant
sin	math.h	non_reentrant
tan	math.h	non_reentrant
acos	math.h	non_reentrant
asin	math.h	non_reentrant
atan	math.h	non_reentrant
atan2	math.h	non_reentrant
cosh	math.h	non_reentrant
sinh	math.h	non_reentrant
tanh	math.h	non_reentrant
ceil	math.h	non_reentrant
floor	math.h	non_reentrant
pow	math.h	non_reentrant
modf	math.h	non_reentrant
fpsave	math.h	reentrant
fprestore	math.h	reentrant
<b>2. 标准输入输出函数</b>		
_getkey	stdio.h	reentrant
getchar	stdio.h	reentrant
gets	stdio.h	non_reentrant
ungetchar	stdio.h	reentrant
putchar	stdio.h	reentrant
printf	stdio.h	non_reentrant
sprintf	stdio.h	non_reentrant
puts	stdio.h	reentrant
scanf	stdio.h	non_reentrant
sscanf	stdio.h	non_reentrant

续表 9.1

函数名	头文件	属性
3. 动态存储函数		
calloc	stdlib.h	non_reentrant
free	stdlib.h	non_reentrant
init_mempool	stdlib.h	non_reentrant
malloc	stdlib.h	non_reentrant
realloc	stdlib.h	non_reentrant
4. 字符归类函数		
isalpha	ctype.h	reentrant
isalnum	ctype.h	reentrant
isctrl	ctype.h	reentrant
isdigit	ctype.h	reentrant
isgraph	ctype.h	reentrant
isprint	ctype.h	reentrant
ispunct	ctype.h	reentrant
islower	ctype.h	reentrant
isupper	ctype.h	reentrant
isspace	ctype.h	reentrant
isxdigit	ctype.h	reentrant
toascii	ctype.h	reentrant
toint	ctype.h	reentrant
tolower	ctype.h	reentrant
_tolower	ctype.h	reentrant
toupper	ctype.h	reentrant
_toupper	ctype.h	reentrant
5. 字符串函数		
memchr	string.h	reentrant/intrinsic
memcmp	string.h	reentrant/intrinsic
memcpy	string.h	reentrant/intrinsic
memccpy	string.h	non_reentrant
memmove	string.h	reentrant/intrinsic
memset	string.h	reentrant/intrinsic
strcat	string.h	non_reentrant
strncat	string.h	non_reentrant
strcmp	string.h	reentrant/intrinsic
strncmp	string.h	non_reentrant
strcpy	string.h	reentrant/intrinsic
strncpy	string.h	non_reentrant
strlen	string.h	reentrant
strchr	string.h	reentrant
strrchr	string.h	reentrant
strrpos	string.h	reentrant



续表 9.1

函数名	头文件	属性
strspn	string.h	non_reentrant
strespn	string.h	non_reentrant
strpbrk	string.h	non_reentrant
strrpbrk	string.h	non_reentrant
6. 字符串转换函数		
atof	stdlib.h	reentrant
atol	stdlib.h	reentrant
atoi	stdlib.h	reentrant
7. 参数函数		
va_start	stdarg.h	reentrant
va_arg	stdarg.h	reentrant
va_end	stdarg.h	reentrant
8. 全程跳转函数		
setjmp	setjmp.h	reentrant
longjmp	setjmp.h	reentrant
9. 内部函数		
_crol_	intrins.h	reentrant/intrinsic
_cror_	intrins.h	reentrant/intrinsic
_irol_	intrins.h	reentrant/intrinsic
_iror_	intrins.h	reentrant/intrinsic
_lrol_	intrins.h	reentrant/intrinsic
_lor_	intrins.h	reentrant/intrinsic
_nop_	intrins.h	reentrant/intrinsic
_testbit_	intrins.h	reentrant/intrinsic
10. 抽象数组		
CBYTE	absacc.h	reentrant
DBYTE	absacc.h	reentrant
PBYTE	absacc.h	reentrant
XBYTE	absacc.h	reentrant
CWORD	absacc.h	reentrant
DWORD	absacc.h	reentrant
PWORD	absacc.h	reentrant
XWORD	absacc.h	reentrant

## 第十章 C51 SFR 头文件和配置文件

超星阅读器  
使用本复制品  
请尊重相关知识产权!

### 10.1 特殊功能寄存器头文件

8051 系列微控器的片上 SFR(特殊功能寄存器)特别多,厂家为介绍上的方便,给它们用预定义的关键字都起了名。在 C51 中,希望沿用这些厂家的命名作为 SFR 的标识符,以便以后可以像已定义过的变量标识符一样,用 SFR 的预定义关键字直接存取 SFR 的内容。为了做到这一点,在 C 语言中,是要加以说明的。按照 C 语言的习惯,把一种芯片上的全部 SFR,包括可按位寻址的位都用厂家已命名的关键字重新说明成标识符。并且,全部集中放在一个叫做特殊功能寄存器的头文件中。以便以后在程序中用预编译伪指令 `#include` 把它包含到源文件之中。在程序中使用任何一个 SFR(包括位 SFR)的预定义关键字就不需再做说明了。随 C51 软件一起,厂家提供了一些常用芯片的 SFR 头文件,如 `REG51.H`, `REG52.H`, `REG51F.H`, `REG515.H` 等。它们按习惯放在 C51 的 `INCLUDE` 目录下。

凡软件商未提供的寄存器定义文件的,用户都可以自己编写。定义 SFR 和位 SFR 的方法见 7.1.2 和 7.1.3。

寄存器定义文件可以自由起名,但扩展名要用 `.h(.H)`。起名最好具有鲜明的可读性。习惯上用下面两种形式:

`REGxxx.H` 或 芯片型号.H

其中:

`xxx` 为芯片型号的缩写。如:

`REG52.H` 或 `80C552.H`

`REG51F.H` 或 `80C750.H`

用 `#include` 伪指令将寄存器头文件包含于源文件有两种写法,即:

`#include <标准寄存器头文件>`

`#include "自编寄存器头文件"`

所谓标准头文件是指厂家提供的,并且已经放在 `include` 目录下的文件。自编的头文件一般是在当前目录下或源文件所在目录下的。二种写法的本质区别在于预编译器搜索文件的路径不同,在 C 语言基础部分的预处理器一章中已作介绍。

### 10.2 C51 配置文件

本节是写给高级程序员的。初学读者当前不必读,以免浪费时间。

C51 编译器根据具体芯片的硬件环境有四种配置用的文件可能需要修改。它们是:

- `STARTUP.A51`——C51 编译器的启动程序部分。它对栈和存储器进行初始化。
- `INIT.A51`——本文件对系统做进一步地补充初始化。

·PUTCHAR.C——本文件是低层输出设备的驱动程序。输出设备不是串行口时,即应修改。如使用 LED 等时即应修改。

·GETKEY.C——本文件是低层输入设备的驱动程序。输入设备不是串行口时,即应修改。如使用矩阵小键盘时即应修改。

上述文件,前两个是汇编程序,后两个是 C 语言程序。上述四个文件直接包含在 C 运行时间库中,连接时是自动进行的,不需要显式的指明。只有当对上述文件做过修改,并重新汇编或编译之后,在连接时应显式地在命令行上增加它们的目标文件,但是,运行时间库千万不要改动。只要把修改过的目标文件在库的前面输入,库中对应的文件就会自动被忽略。

### 10.2.1 STARTUP.A51 文件

本文件为汇编源文件,其开始 EQU 说明部分设定如下:

·IDATALEN——是开机时需对片内 RAM 初始化为 0 的长度。对 8051 默认值为 80H, 8052 为 100H。对于掉电模式欲保留原系统状态时,需抑制用 0 做初始化。这时, IDATALEN 应置为 0。

·XDATASTART 和 XDATALEN——它们是 XDATA 空间需要用 0 初始化的首地址和长度。

·PDATASTART 和 PDATA——它们是 PDATA 空间需要用 0 初始化的首地址和长度。

·IBPSTACK 和 IBPSTACKTOP——它们为 small 模式指定重入栈。IBPSTACK 表明是否应对栈指针(? C\_IBP)进行初始化。IBPSTACKTOP 指明栈顶地址。片内 RAM 为 256 字节时,而用 0xFF 作为栈顶时可不对栈指针(? C\_IBP)进行初始化。C51 不对重入栈是否满足用户特定的需要做检查,用户必须自己进行测试。

·XBPSTACK 和 XBPSTACKTOP——它们为 large 模式指定重入栈。意义与 small 模式重入栈相似。XBPSTACK 表明是否应对栈指针(? C\_XBP)进行初始化。XBPSTACKTOP 指明栈顶地址。用 XDATA 空间的 0xFFFF 作栈顶地址时,可不对栈指针(? C\_XBP)进行初始化。C51 不对重入栈是否满足用户特定的需要做检查,用户必须自己进行测试。

·PBPSTACK 和 PBPSTACKTOP——它们为 compact 模式指定重入栈。意义与 small 模式重入栈相似。PBPSTACK 表明是否应对栈指针(? C\_PBP)进行初始化。PBPSTACKTOP 指明栈顶地址。用 PDATA 空间的 0xFF 作栈顶地址时,可不对栈指针(? C\_PBP)进行初始化。C51 不对重入栈是否满足用户特定的需要做检查,用户必须自己进行测试。

·PPAGEENABLE 和 PPAGE——它们用于 compact 和 large 模式。PPAGEENABLE 开放对 P2 口的初始化。PPAGE 将使 P2 口置为 XDATA 空间的 256 页中的一页。

[例 10.1]

```
PPAGEENABLE = 1, PPAGE = 10
```

则将使 P2 口置为指定的页码 10。页空间的首地址 PDATA 的定位范围在 1 000H(10 页)到 10FFH 之间。L51 行命令会用到首地址 PDATA。它不应与 P2 口所置的页码相矛盾。large 模式的 XDATA 空间也可分页使用,以提高存取效率。

### 10.2.2 INIT.A51 文件

本文件是系统初始化的补充程序。一些放在 startup.a51 中不大合适的初始化内容放在

这里。经常把看门狗的初始化程序放在这里。如果系统初始化时间大于程序中看门狗定时,更新时间就应该在这里禁止看门狗,或者延长定时时间。看门狗的定时时间写成宏的形式,用时最为方便。需要刷新的地方只需写宏名即可。

使用本复制品  
请尊重相关知识产权!

### 10.2.3 PUTCHAR.C 文件

文件 PUTCHAR.C 包含的是输出设备的低层驱动程序。当前输出设备为串行口时,采用 XON/XOFF 通讯协议,换行字符 LF(\n)被转为 CR 和 LF。这是为满足一些终端的需要。用户如果改用其他输出设备(如 LCD、LED 等)时,应改写这个文件的程序。

### 10.2.4 GETKEY.C 文件

文件 GETKEY.C 包含字符输入设备的低层驱动程序。当前输入设备用的是串行口读入一个字符的驱动程序。由它供给库函数 getchar()使用。用户如果改用其他输入设备(如矩阵键盘等)时,应改写本文件的程序。

## 第十一章 C51 预处理器伪指令

C51 的预处理器伪指令与基础部分介绍的预处理器伪指令基本上一样, 所以不再介绍。仅有一项应加以说明, 就是预处理器的预定义宏。C51 预定义的宏如下:

• 与基础部分一致的有:

__ TIME __	__ DATE __
__ FILE __	__ LINE __
__ STDC __	

• 新定义的有:

__ C51 __	__ MODEL __
-----------	-------------

• 它们的意义如下:

__ TIME __	开始编译时间。
__ DATE __	编译日期。
__ FILE __	编译文件名。
__ LINE __	编译文件的当前行。
__ STDC __	与 ANSI C 兼容时恒为 1。
__ C51 __	C51 的版本号, 如 3.00 则给出 300。
__ MODEL __	存储模式: 0 为 small; 1 为 compact; 2 为 large。

## 第十二章 C51 编译命令行控制 选项和控制伪指令

### 12.1 简介

C 语言编写的源程序必须用 C51 命令行的形式进行编译。编译所生成的可再定位的目标文件,再用连接器 L51 连接定位,生成可执行的绝对目标程序。绝对目标程序经调试器调试无误后,用编程器写入 8751 微控制器的 EPROM 中去。微控制器一经复位即开始运行。

本章主要介绍如何对 C 源文件进行编译。

### 12.2 编译命令行

用命令行编译源程序的格式如下:

C51 源文件名[控制选项表]

其中:

(1) C51——编译器名。

(2) 源文件名——以 .c 为扩展名的 C 语言源文件。

(3) [控制选项表]——以空格符分隔的多个控制选项,也可以只有一个控制选项或没有。

控制选项是引导编译器怎样对源文件进行编译的。C51 编译器的各种控制选项列于表 12.1。

(4) 源文件被编译后,将生成目标文件。如果需要,还可以通过控制选项生成列表文件。

(5) 控制选项有三类(参见表 12.1):

- 作用于源文件之上的,决定如何选用源文件;
- 作用于目标文件的,决定产生怎样的目标文件;
- 作用于列表文件,决定产生怎样的列表文件。

(6) 命令行上使用的控制选项,多可以加上预处理器伪指令 #pragma,放在源文件中。这时,叫做编译控制伪指令。它们虽写在源文件之中,但却在编译时起控制作用。其作用与命令行控制选项一样。

(7) 同样的控制选项如果两处都有,则以命令行上的为准。

(8) 控制伪指令在源文件中使用,有的只能使用一次,有的则可多次使用。作用正好相反的伪指令可以解除其作用。它们在表 12.1 中都得到反映。

下面各节对表 12.1 中的各项,逐一地进行介绍。介绍时按编译伪指令一次性使用的和可多次使用的加以分类。

### 12.2.1 一次性使用编译控制伪指令

#### 1. DEFINE(DF)

本选项只能用于命令行。在命令行上定义一些名字,这些名字应与源程序中的条件编译伪指令相呼应,以便选定条件编译中的哪一个编译条件。名字是大小写敏感的,使用时应予以注意。

命令行格式:

C51 源文件名 DEFINE(名表)

其中:

名表用作条件编译中的条件。名表是用逗号分隔的多个名字以满足多个条件编译的需要。

表 12.1 编译器命令行控制选项和控制伪指令

控制对象	控制选项和控制伪指令	缺省值	缩写
一次使用	源文件	DEFINE	-
		[NO]EXTEND	EXTEND
	目标码	[NO]DEBUG	NODEBUG
		[NO]OBJECT	OBJECT(xxx.obj)
		OPTIMIZE	OBTIMIZE(5, SPEED)
		SMALL COMPACT LARGE	SMALL
		[NO]INTVECTOR	INTVECTOR
		[NO]OBJECTEXTEND	NOOBJECTED
		ROM	ROM(LARGE)
			-
	列表文件	[NO]LISTINCLUDE	NOLISTINCLUDE
		[NO]SYMBOLS	NOSYMBOLS
		[NO]PREPRINT	NOPREPRINT
		[NO]CODE	NOCODE
		[NO]PRINT	PRINT
		[NO]COND	COND
		PAGELength	PAGELength(69)
		PAGEWIDTH	PAGEWIDTH(132)
多次使用	源文件	SAVE	-
		RESTOR	-
		DISABLE	-
	目标码	[NO]REGPARMS	REGPARMS
		REGISTERBANK(0~3)	REGISTERBANK(0)
		[NO]AREGS	AREGS
	列表文件	EJECT	EJ

[例 12.1] 条件编译示例。

在源文件 sample.c 中有:

```

...
#if check
    ...
    /* 检查用语句 */

```



```
# endif
...
# if NoExtRam
    s2: ... /* 无外部 RAM 时编译 */
# elif
    s3: ... /* 有外部 RAM 时编译 */
# enclif
...
```

在编译命令行中:

C51sampl.cDEFINE(check, NoExtRam)

由于命令行中, DEFINE 名表中定义了 check 和 NoExtRam, 从而在预处理时选入了 s<sub>1</sub> 段和 s<sub>2</sub> 段进行编译, s<sub>3</sub> 段则不被编译。

## 2. [No]EXTEND

说明:指示 C51 对源文件中的非 ANSI C 扩展部分是否也预以编译。

[例 12.2] 对非 ANSI C 部分不做编译。有:

```
C51 SAMPLE.C NOEXTEND
或者 #Pragma NOEXTEND
```

## 3. [NO]DEBUG

说明:指示目标文件中是否包含调试信息。带有 DEBUG 选项生成的可执行目标程序, 支持在仿真器上做符号调试。

## 4. [NO]OBJECT

格式:

OBJECT(xxx.obj)

或 NOOBJECT

说明:xxx.obj 分是用户指定作为目标文件的名称。缺省时使用被编译的源文件主名加 .obj 扩展部。NOOBJECT 指定不生成目标文件。

[例 12.3] C51 SAMPLE.C NOOBJECT

[例 12.4] #pragma OJ (smapple.obj)

## 5. OPTIMIZE

格式:

OPTIMIZE(n) [ SPEED|SIZE ]

其中:

(1) 优化级别 n——C51 提供 0~5 共 6 级优化。缺省级别为 5。

•OPTIMIZE(0):

常数(包括地址常量)因子化。

8051 内部 RAM 及位访问优化。

跳转优化到绝对目标。

•OPTIMIZE(1):

死码消除。

条件跳转细查, 简化或消除。

超星浏览器提醒您:  
使用本复制品  
请尊重相关知识产权!

## ·OPTIMIZE(2):

数据覆盖:鉴别静态可覆盖的变量,包括位变量并加以标注。

由 C51 对全局数据进行分析,选取可静态覆盖段。

## ·OPTIMIZE(3):

聚焦优化:冗余 MOV 指令删除(包括不必要的寄存器和常量向内存存取)。

只要能节省存储空间和执行时间的,复杂操作用简单操作代替。

## ·OPTIMIZE(4):

参数和自动变量要尽可能地安排为寄存器变量。

间接存取的变量要直接装入间址寄存器,不要经过中间寄存器(涉及 idata, pdata, xdata, code 空间)。

局部公有子式因子化。第一次计算得到的结果留待以后套用,消除代码中的复杂计算。

switch/case 语句向跳转表或跳转串优化。

## ·OPTIMIZE(5):

全局公用子式因子化,并将中间结果放寄存器中。

包括 0~4 级优化。

(2) [speed|size]——指明优化的侧重点是执行速度还是代码长度。什么也不指定的缺省值是速度。

注:全局性优化时需要较多的内存,不足时会给出错信息。

[例 12.5] C51 sample.c OPTIMIZE(4)

[例 12.6] #Pragma ot(4, size)

6. small(SM), compact(CP), large(LA)

说明:它们控制编译时采用的存储模式。模式影响参数和局部数据的存储空间。

[例 12.7] C51 SAMPLE.C LARGE

[例 12.8] #pragma compact

7. [NO]INTVECTOR

说明:是否由编译器自动代为填写中断向量的表项。

[例 12.9] C51 SAMPLE.C NOIV

[例 12.10] #pragma nointvector

8. [NO]OBJECTEXTEND

说明:OBJECTEXTEND 指示编译器在产生目标文件时,包括附加变量类型的定义信息。由此产生的目标文件是 intelOMF-51 的超集。与它配套的定位器 L51 应选用 2.30 以上的版本。能够调试它的仿真器和模拟器应具备增强目标文件装载器。遇仿真器不能适应时,应改成 NOOBJECTEXTEND 控制选项,再行编译试之。

[例 12.11] C51 SAMPLE.C DEBUG OBEJECTEXTEND

[例 12.12] #pragma db oe

9. ROM(SMALL|COMPACT|LARGE)

说明:规定程序内存的大小、函数指针的长度和跳转的空间大小。

(1) ROM(SAMLL)——code 空间最大 2K。函数长度最大 2K。调用和跳转指令分别使

用 ACALL 和 AJMP。

(2) ROM(COMPACT)——code 空间最大 64K。函数长度最大 2K, 故函数内跳转用 AJMP。调用指令使用 LCALL。

(3) ROM(LARGE)——code 空间最大 64K。函数长度最大 64K。调用和跳转指令分别使用 LCALL 和 LJMP。

[例 12.13] C51 SAMPLE.C ROM(SMALL)

[例 12.14] #pragma ROM(SMALL)

#### 10. [NO]LISTINCLUDE

说明: 指示是否将头文件内容加入列表文件中。缺省为 NOLISTINCLUDE。

[例 12.15] C51 SAMPLE.C LISTINCLUDE

[例 12.16] #pragma listinclude

#### 11. [NO]SYMBOLS

说明: 指示是否产生本模块使用的符号表列表文件。符号表包括变量和函数的符号名、类型、存储类、存储空间、偏移地址和占用内存大小。

[例 12.17] C51 SAMPLE.C SYMBOLS

[例 12.18] #pragma symbols

产生的列表文件内容(一部分)如下:

NAME	CLASS	MSPACE	TYPE	OFFSET	SIZE
EA .....	ABSBIT	—	BIT	00AFH	1
update PUBLIC .....	CODE	PROC	—	—	—
dtime .....	PARAM	DATA	PTR	0000H	3
settime PUBLIC .....	CODE	PROC	—	—	—
mode .....	PARAM	DATA	PTR	0000H	3
dtime .....	PARAM	DATA	PTR	0003H	3
setuptime .....	AUTO	DATA	STRUCT	0006H	3
time .....	* TAG *	—	STRUCT	—	3
hour .....	MEMBER	DATA	U_CHAR	0000H	1
min .....	MEMBER	DATA	U_CHAR	0001H	1
sec .....	MEMBER	DATA	U_CHAR	0002H	1
SBUF .....	SFR	DATA	U_CHAR	0099H	1
ring .....	PUBLIC	DATA	BIT	0001H	1
SCON .....	SFR	DATA	U_CHAR	0098H	1
TMOD .....	SFR	DATA	U_CHAR	0089H	1
TCON .....	SFR	DATA	U_CHAR	0088H	1
mau .....	PUBLIC	CODE	ARRAY	00FDH	119

#### 12. [NO]PREPRINT[(预处理器列表名)]

说明:

(1) 指示是否生成预处理器列表文件。

(2) (列表文件名)——可有可无。列表文件名缺省时, 用源文件主名加扩展名 .l。

[例 12.19] C51 SAMPLE.C PREPRINT

[例 12.20] C51 SAMPLE. CPREPRINT(preprogram.lst)

### 13. [NO]CODE

说明:指示是否在列表文件中对 C 语句的各函数附加对应的汇编助记符指令码。

[例 12.21] C51 SAMPLE. CCODE

[例 12.22] #pragma code

下面给出加过 CODE 控制选项产生的列表文件(片段)。列表文件中带(R)的指令码是可再定位目标码;带(E)的指令码说明码中含有外部变量。它们的真实码在连接时添入。

stmt level source

```

1      extern  unsigned char a, b ;
2              unsigned char c ;
3
4      void    main( )
5      {
6          1      c = 14 + 15 * ((b/c) + 222) ;
7          1      |

```

#### ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION main (BEGIN)
; SOURCE LINE # 5
; SOURCE LINE # 6

0000 E500      E      MOV    A, b
0002 8500F0    R      MOV    B, c
0005 B4        DIV    AB
0006 75F00F    MOV    B, #0FH
0009 A4        MUL    AB
000A 24D2      ADD    A, #0D2H
000C F500 R    MOV    c, A
; SOURCE LINE # 7

000E 22        RET
; FUNCTION main (END)

```

### 14. [NO]PRINT[(列表文件名)]

说明:

(1) 指示是否产生列表文件。

(2) (列表文件名)——由用户指定。它可有可无。列表文件名缺省时,用源文件主名加扩展名 .LST。

[例 12.23] C51 SAMPLE.C PRINT

[例 12.24] #pragma pr( \ user \ list \ mysample.lst)

### 15. [NO]COND

说明:COND 指示条件编译中未被选中的部分是否仍列入列表文件之中。NOCOND 则将条件编译中未被选中的部分干脆弃掉。当选项为 COND 时,虽将未被选中的部分仍列入列表文件之中,但并不给预行号和其他说明。

## [例 12.25] C51 SAMPLE.C COND

产生的列表文件如下(未被选中的 VAX 部分虽列出,但未给行号和其他说明)。

DOS C51 COMPILE V2.0, COMPILATION OF MODULE SAMPLE

OBJECT MODULE PLACED IN SAMPLE.OBJ

COMPILER INVOKED BY: C51 SAMPLE.C COND

stmt	level	source
1		extern unsigned char a, b ;
2		unsigned char c ;
3		
4		void main( )
5		{
6	1	# if defined (VAX)
		c = 13 ;
		# elif defined ( __ time __ )
9	1	b = 14 ;
10	1	a = 22 ;
11	1	# endif
12	1	}



## [例 12.26] C51 SAMPLE.C NOCOND

产生的列表文件如下(未被选中的 VAX 部分未列出):

DOS C51 COMPILE V2.0, COMPILATION OF MODULE SAMPLE

OBJECT MODULE PLACED IN SAMPLE.OBJ

COMPILER INVOKED BY: C51 SAMPLE.C NOCOND

stmt	level	source
1		extern unsigned char a, b ;
2		unsigned char c ;
3		
4		void main( )
5		{
6	1	# if defined (VAX)
		# elif defined ( __ time __ )
9	1	b = 14 ;
10	1	a = 22 ;
11	1	# endif
12	1	}

## 16. PAGEDLENGTH[(0~65 535)]

说明:设置列表文件每页打印的行数。可选值在 0~65 535 中任选。缺省时为 69 行。

[例 12.27] C51 SAMPLE.C PAGEDLENGTH(68)

## 17. PAGEWIDTH[(78~132)]

说明:设置列表文件每行打印的字数。可选值在 78~132 中任选。缺省值为 132 行。

[例 12.28] C51 SAMPLE.C PAGEWIDTH(79)

[例 12.29] #pragma pw(79)



## 12.2.2 可多次使用编译控制伪指令

## 1. [NO]REGPARMS

说明:

(1) 指示其后的函数是否使用寄存器传送参数。

(2) NOREGPARMS 时,禁止使用寄存器传送参数。这时候,参数是放在与编译模式相关的缺省存储空间中传送的。

(3) REGPARMS 使用寄存器传送前三个参数。寄存器不足和三个以上的参数放在与编译模式相关的缺省存储空间中传送。

(4) 它们可在程序中多次使用,以实现有的函数用寄存器传送,有的函数不用。

(5) 注意对库函数和汇编语言函数不要使用本控制伪指令。

(6) 本控制伪指令只能用于源程序中,不能用于编译命令行。

## 2. [NO]AREGS

说明:

(1) 指示对寄存器是否使用绝对地址存取。寄存器绝对地址存取效率较高(PUSH, POP 指令用的都是绝对地址)。

(2) AREGS 指示对寄存器使用绝对地址存取,NOAREG 则解除。在解除寄存器绝对地址存取的情况下,函数是独立于寄存器组的。即它可用任一寄存器组。当前究竟用的是那个寄存器组,要向前追溯 REGISTERBANK 控制伪指令的设定。一般是不必介意当前使用的是那个寄存器组,如果一定要用某一组时,可用 REGISTERBANK 控制伪指令来指定。

(3) 注意:

·本控制伪指令可多次使用。

·本控制伪指令只能用于函数之外,不能用于函数之内。

[例 12.30] C51 SAMPLE.C NOAREGS

[例 12.31] #pragma AREGS

[例 12.32] 在源文件中 NOAREGS 与 AREGS 交叉使用示例。

stmt	level	source
1		extern char func( );
2		chara ;
3		
4		#pragma NOAREGS

```

5          noaregfunc( )|
6      1          a = func( ) + func( ) ;
7      1          |
8
9          #pragma AREGS
10         aregfunc( )|
11      1          a = func( ) + func( ) ;
12      1          |

```



## ASSEMBLY LISTING OF GENERATED OBJECT CODE

```

; FUNCTION noaregfunc (BEGIN)
; SOURCE LINE # 6

0000 120000    E    LCALL func
0003 EF R      MOV  A,R7
0004 C0E0      PUSH ACC
0006 120000    E    LCALL func
0009 D0E0      POP  ACC
000B 2F        ADD  A,R7
000C F500 R    MOV  a,A
; SOURCELINE# 7

000E 22        RET
; FUNCTION noaregfunc (END)
; FUNCTION aregfunc (BEGIN)
; SOURCE LINE # 11

0000 120000    E    LCALL func
0003 C007 R    PUSH AR7
0005 120000    LCALL func
0008 D0E0 E    POP  ACC
000A 2F        ADD  A,R7
000B F500 R    MOV  a,A
; SOURCE LINE # 12

000D 22        RET
; FUNCTION aregfunc (END)

```

## 3. SAVE/RESTORE

说明:SAVE 把当前设置的 REGPARMS、AREGS 和 OPTIMIZE(n)[speed|size]推入堆栈,保护起来。RESTORE 则由栈中弹出,恢复它们。

注意:

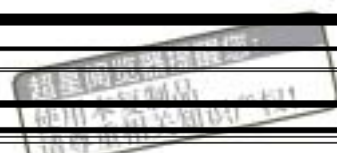
- (1) 本控制伪指令只能用于源程序中,不能用于编译命令行上。
- (2) 本控制伪指令可以多次使用。

[例 12.33] 禁止两个外部函数用寄存器参数传送。

```
#pragma save
```



```
#pragma noregparms
```



```

0005          ? C0002;
0005 C0D0      PUSH      PSW
; - - - - variable 'p1' assigned to register 'R7' - - - -
; - - - - variable 'p2' assigned to register 'R5' - - - -
; SOURCE LINE # 4
; SOURCE LINE # 5

0007 ED        MOV      A, R5
0008 8FF0      MOV      B, R7
000A A4        MUL      AB
000B 25E0      ADD      A, ACC
000D FF        MOV      R7, A
; SOURCE LINE # 6

000E          ? C0001;
000E D0D0      POP       PSW
0010 92AF      MOV      EA, C
0012 22        RET
; FUNCTION _ dfunc (END)

```



## 6. EJECT

说明：

- (1) 使列表文件换页。
- (2) 只能用于源程序, 不能用于编译命令行。

[例 12.37] #pragma eject

## 第十三章 C51 及 L51 使用方法

### 13.1 C51 的使用环境

硬件要求:

- IBM \_ PC/XT/AT/386/486/pendium 及其兼容机。
- 384K 字节以上内存。
- 一个软盘驱动器和一个硬盘驱动器。

软件要求:

- MS \_ DOS 或 PC \_ DOS 3.0 版本以上。



### 13.2 C51 安装

1. 在 autoexec.bat 文件中加入下列命令行:

- SET PATH = 驱动器名:\路径名  
(如:SET PATH = C:\C51\BIN)
- SET C51LIB = 驱动器名:\路径名  
(如:SET C51LIB = C:\C51\LIB)
- SET TEMP = 驱动器名:\路径名  
(如:SET TEMP = C:\ )
- SET C51INC = 驱动器名:\路径名  
(如:SET C51INC = C:\C51\INCLIB,C:\C51\CHIP\_DIR)

注:搜索路径最多给 10 个,并用逗号分隔。

2. 在 config.sys 文件中,加入或改写成如下命令:

- files=20
- buffers=20
- shell=驱动器名:\路径名\command.com/e;xxx/p

(仅在 config.sys 文件中,SET 命令所加入环境变量过多,超过了预留缓冲区容量时,才增加本行。如:shell=C:\command.com/e;512/p)

3. 使 PC 机的控制转向 A 驱动器,插入 C51 软盘,键入:

- A:> install 驱动器名:\路径 <CR>  
(如 install c:\ <CR> )

### 13.3 编译方法

#### 1. 进入安装有 C51 的驱动器

驱动器名: <CR>

(如 C:<CR>)

#### 2. 进入装有 C51 的目录,使其成为当前目录

CD 装有 C51 的路径名 <CR>

(如 CD \ C51<CR>)

#### 3. 键入编译命令行

C:\C51> C51 源文件名 [控制选项表]

其中:

(1) 源文件名一般是具有扩展名为 .C 的文件。

(2) 控制选项表是用空格分隔的编译控制选项。控制选项表可有可无。没有时,意味着使用控制选项的缺省值,或在源程序中有等价于选项的预处理控制伪指令行。

[例 13.1]

C:\C51> C51 SAMPLE.C DEBUG CODE SYMBOLS OBJTEXTEND

在当前目录中生成: SAMPLE.OBJ 目标文件和 SAMPLE.LST 列表文件。

[例 13.2]

C:\C51> C51 SAMPLE.C NOOBJECT PREPRINT

在当前目录中只生成预处理后的源文件 SAMPLE.L

### 13.4 C51 支持的文件名和设备名

·支持与 DOS 相同的文件名。即全文件名为:驱动器名:\路径名\最多 8 字符文件名。最多 3 字符扩展名。

·支持控制台作为缺省的标准输入输出设备。控制台用 DOS 关键字 CON 表示,也可沿用 IntelISIS 开发系统使用的:CO:表示。

·空文件用 DOS 关键字 NUL 表示,也可沿用 IntelISIS 开发系统使用的:BB:表示。

·打印机用 DOS 关键字 PRN 表示,也可沿用 IntelISIS 开发系统使用的:LP:表示。

### 13.5 错误号

在 PC 机 DOS 操作系统下运行的 C51BIN 文件,在运行结束时返回的状态码放在 DOS 环境变量 ERRORLEVEL 中。ERRORLEVEL 所对应的出错原因有:

0:无错,无警告。

1:有警告。

2:有错。

3:有致命错。



ERROR LEVEL 可用于 DOS 批处理文件中作为 IF 分支条件。

## 13.6 连接/定位方法

L51 是与 C51 配套的连接/定位器。

图 13.1 是用 C、汇编、PL/M 多种语言混合开发应用程序的过程示意图。由其中可了解到 L51 所处的地位和作用。简而言之,连接/定位器完成下列操作:

1. 连接——将一个程序的多个源文件(包括多种语言文件)产生的各目标文件与用户自定义库文件(如果有的话)及标准库文件连接起来。

2. 连接的内部规则——是依文件的顺序按段类型(CODE, DATA, PDATA, XDATA 和 BIT 等 5 个函数段类型)把不同函数模块归类连接为多个整段。各段类型的整段当前还是可再定位段。段类型中带 OVERLAYABLE 属性的是要另外分别归类的。所以,由 OVERLAYABLE 属性的各段类型组成的各整段是可覆盖的可再定位段。归类连接的接续方式有:PAGE, INPAGE, BLOCK, BITADDRESSABLE 和 UNIT 等 6 种。各种段类型的整段不应超过各自对应空间的可能长度。

3. 连接时很重要的细部工作有:

- 把引用和外部变量的定义说明一个一个地匹配起来,把函数调用和函数的定义说明一个一个地匹配起来,检查他们的正确性,并归纳出交叉访问表。

- 对函数用绝对地址调用的,对用函数指针做参数而又未指定 reentrant 的函数,都要仔细检查函数之间是否存在有直接或间接的递归调用。不存在的,按覆盖处理。存在的则告警,使程序员能够通过 L51 命令行上增加 OVERLAY 的特殊连接控制,进行人工干预,消除 L51 可能的对递归调用的误解。

4. 绝对定位——首先把 C 语言和汇编语言中的绝对段定位,其次按 L51 命令行中关于各存储空间分配的强制定位意见进行定位,然后从各自剩余空间的最低可能位置定位各可再定位的段类型整段,最后定位可再定位的 OVERLAYABLE 段类型整段加上编译/考虑到,将各可定位整段。

5. ?STACK 段——最后安排?STACK 段。它充满存储模式缺省空间的剩余部分。它自下而上地堆放。使用片上 RAM 做堆栈时,即使用的是 DATA 空间也要换用 IDATA 空间,以便间接寻址。

6. 定位到绝对地址——在把数据和程序定位到绝对地址上之后,便要通过计算把可再定位程序段中的再定位部分和外部变量部分置换到正确的绝对值上来。

7. 两个文件——L51 最后生成两个文件:绝对目标文件和映象文件。前者就是可执行的绝对目标程序,后者是提交给程序员的最终各空间映像表。

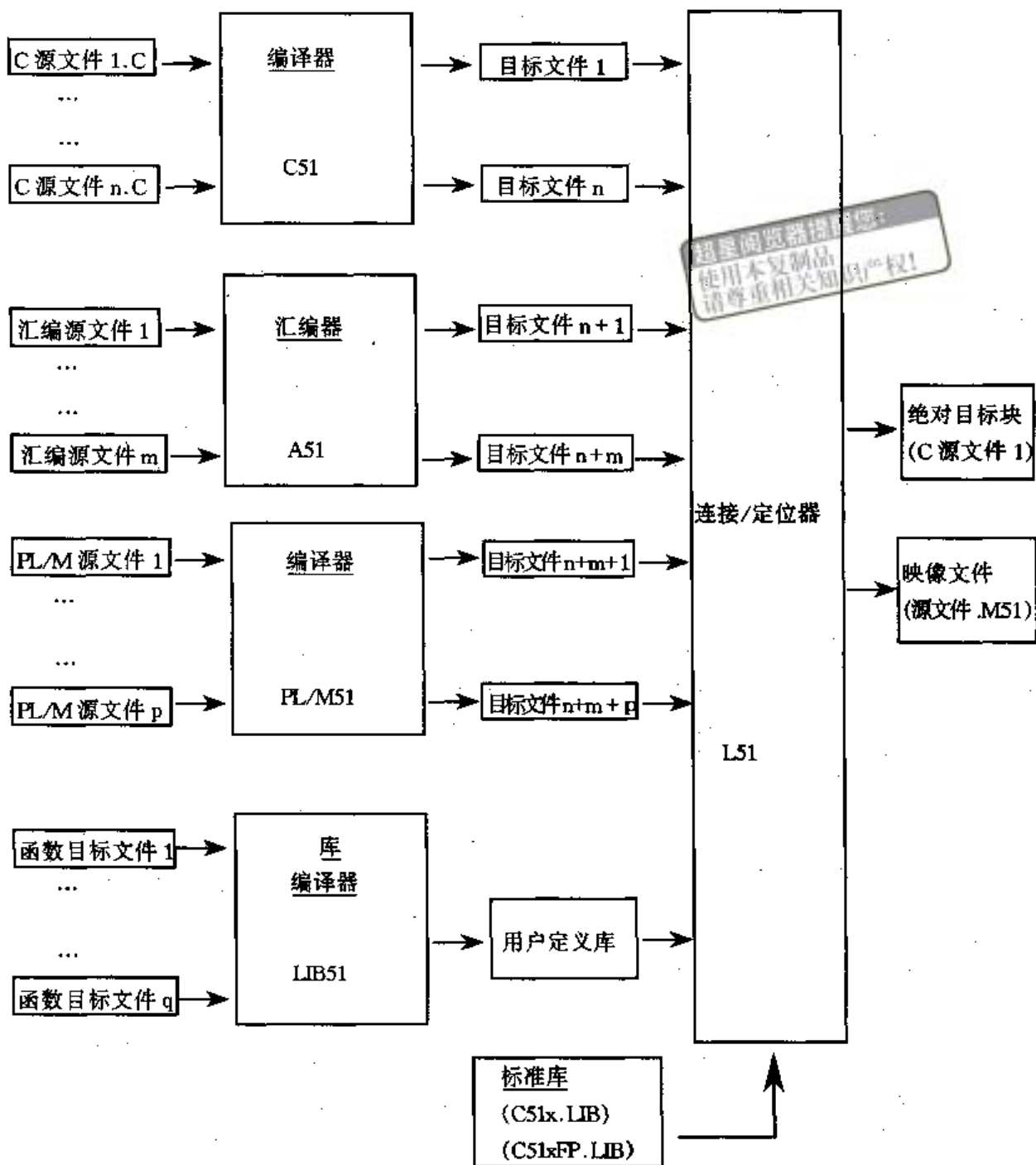


图 13.1 程序的混合语句开发流程链

## 13.7 连接控制选项

### 13.7.1 一般的连接控制选项

表 13.1 给出连接控制的一般选项。

表 13.1 一般连接控制选项

控制选项	作用
NAME(模块名)	用 NAME 的参数部分指定绝对目标模块名,缺省时以主目标文件的主名为模块名,同时也是绝对目标文件名
[NO]DEBUGSYMBOLS	指定绝对目标模块中是否带有调试用符号表
[NO]DEBUGPUBLICS	指定绝对目标模块中是否带有调试用公用符号表
[NO]DEBUGLINES	指定绝对目标模块中是否带有调试用的行号

注:表中各选项的缺省值,均为其肯定值。一般只在不需要调试用的 SYMBOLS、PUBLICS 和 LINES 时才使用选项。

### 13.7.2 特殊的连接控制选项

特殊连接控制选项主要有两个:

- 运行库自动连接解除;
- 函数覆盖检查遇到难点时的程序员引导。

下面分述之。

#### 1. NODEFAULTLIBRARY

关闭运行时间库的自动连接。一般是不应该关闭的。其所以要关闭是打算用命令中显式给出的库替换原来的运行时间库(例如用 PL/M51 库代替 C51 库)。

#### 2. NOOVERLAY——连接时禁止覆盖。

#### 3. OVERLAY(参数)——按参数要求进行覆盖。

其中:

(参数)是由下列四种基本形式和它们的任意组合构成。参数的四种基本形式:

(1) (函数段名 1 ! (函数段名 2 [, 函数段名 3] ...))——表示函数段名 1 可与其后指明的函数段相覆盖。

(2) (函数段名 1 ~ (函数段名 2 [, 函数段名 3] ...))——表示函数段名 1 不可与其后指明的函数段相覆盖。

(3) (函数段名 1 ! \* )——表示函数段名 1 可与任何其他函数段相覆盖。

(4) (函数段名 1 ~ \* )——表示函数段名 1 不可与任何其他函数段相覆盖。

在 C51 中,是假定除 CODE 空间外,每个函数的数据空间都是可与其他函数段相覆盖的。所以,在内部为每个函数定义了一个段名,并赋预了存储空间属性和 OVERLAYABLE 的连接属性。这样定义的段有下面四类:

函数段	存储空间属性	连接属性
? BI ? 函数名? 模块名	BIT	OVERLAYABLE
? DT? 函数名? 模块名	DATA	OVERLAYABLE
? PD? 函数名? 模块名	PDATA	OVERLAYABLE
? XD? 函数名? 模块名	XDATA	OVERLAYABLE

C51 既然假定函数的数据空间都是 OVERLAYABLE,它就要保证每个函数的参数和局部



变量所在的空间确实可以覆盖,在运行中不出问题。为此 C51,按下列规则进行检查,符合的就一定是可覆盖的:

- 相关代码段间不存在直接或间接传址访问。
- 函数只被 main()调用或只被中断函数调用。

但是,自动覆盖分析有两个难点,即:

- 函数之间的间接传址访问。
- main 与中断的混合调用。

遇到这些情况,就要程序员运用上述带参数的 OVERLAY()选项进行正确的导引。具体例见后。



## 13.8 定位控制选项

一般用 C51 编译得到的目标文件,用 L51 自动地去连接和定位,是能够得到满意绝对目标程序的。也就是说,用户无需使用定位控制选项去干预 L51 的定位过程。只有在 L51 自动安排的结果不如人意,或用户有特殊的定位要求时,才会使用控制选项。

下面列出各种定位控制选项的格式:

### 1. RAMSIZE(RAM 字节数)

本选项是考虑 8051 派生芯片太多,有的片上 RAM 可能不标准。这时,使用本控制选项将实际字节数通知 L51。参数部分取值范围为 128~256。缺省值为 128。

### 2. PRECEDE(段名(基地址) [,段名(基地址)]...)

其中:

(1) PRECEDE——定位用存储空间的一种。它专指片上 RAM 细分出来的前区空间(只包括通用寄存器区和位寻址区),地址范围 00H~2FH。

(2) 段名——存储空间属性为 DATA 的函数段名。

(3) 基地址——上述函数段的起始地址。

### 3. DATA(首地址 | 段名(基地址) [,段名(基地址) ...] [,首地址 | 段名(基地址) [,段名(基地址) ...]...)

其中:

(1) DATA——定位存储空间的一种。指片上直接寻址 RAM 空间。地址范围 00H~7FH。

(2) 首地址——在本定位存储空间为后续内容指定绝对地址。缺省值为本定位存储空间的 30H 地址。

(3) 段名——存储空间属性为 DATA 的函数段名。

(4) 基地址——上述函数段的起始地址。

### 4. IDATA(首地址 | 段名(基地址) [,段名(基地址) ...] [,首地址 | 段名(基地址) [,段名(基地址) ...]...)

其中:

(1) IDATA——定位存储空间的一种。指片上间址 RAM 空间。地址范围 00H~FFH。

(2) 首地址——在本定位存储空间为后续内容指定绝对地址。缺省值为本定位存储空间

的 30H 地址。

(3) 段名——存储空间属性为 DATA 的函数段名。

(4) 基地址——上述函数段的起始地址。

5. XDATA(首地址 | 段名(基地址) [, 段名(基地址) ...] [, 首地址 | 段名(基地址) [, 段名(基地址) ...] ...)

其中:

(1) XDATA——定位存储空间的一种, 指片外 RAM 空间。地址范围 0000H~FFFFH。

(2) 首地址——在本定位存储空间为后续内容指定绝对地址。缺省值为本定位存储空间的 0 地址。

(3) 段名——存储空间属性为 XDATA 的函数段名。

(4) 基地址——上述函数段的起始地址。

6. CODE(首地址 | 段名(基地址) [, 段名(基地址) ...] [, 首地址 | 段名(基地址) [, 段名(基地址) ...] ...)

其中:

(1) CODE——定位存储空间的一种, 指片内片外连续的 ROM 空间。地址范围 0000H~FFFFH。

(2) 首地址——在本定位存储空间为后续内容指定绝对地址。缺省值为本定位存储空间的 0 地址。

(3) 段名——存储空间属性为 CODE 的函数段名。

(4) 基地址——上述函数段的起始地址。

7. BIT(首地址 | 段名(基地址) [, 段名(基地址) ...] [, 首地址 | 段名(基地址) [, 段名(基地址) ...] ...)

其中:

(1) BIT——定位存储空间的一种, 指片内 RAM 位空间。地址范围 20H~2FH(位地址范围 0H~7FH)。

(2) 首地址——在本定位存储空间为后续内容指定绝对地址。缺省值为本定位存储空间的 0 地址。

(3) 段名——存储空间属性为 BIT, BITADDRESSABLE, DATA 的函数段名。

(4) 基地址——上述函数段的起始地址。属性为 BITADDRESSABLE 时, 基地址应是可被 8 整除的值, 即一定要在字节界上。

8. PDATA(首地址)

其中:

(1) PDATA——定位存储空间中特殊的一种。它是指片外 RAM 空间 256 页中指定的一页(256 字节)。

(2) 首地址——指定页页首的绝对地址(即能被 100H 所整除)。缺省值为片外 RAM 0 页页首地址。

这种定位存储空间是为了满足 MEDIUM 存储模式和混合存储模式的需要。指定 0 页正是 MEDIUM 模式。指定其他页是为 LARGE 模式指定一个高效页, 使在此页内能够改用效率较高的页内寻址。页内寻址用 MOVX @R0 代替 MOVX @DPTR, 速度可快 25%~

30%。

为 LARGE 模式指定高效页时, 用户必需改写启动文件 STARTUP. A51 中的 PPAGEENABLE 和 PPAGE, 以实现将 P2 口置为页面地址。

#### 9. STACK(段名(基地址) [, 段名(基地址)]...)

其中:

(1) STACK——也是定位存储空间的一种。它总是定位片上间址空间(IDATA)的未用部分。栈底在下。这种定位选项的设立是为了充分利用这一段片上间址空间。本定位选项不否定各存储模式的覆盖堆栈。

(2) 段名——存储空间属性为 STACK 的段名。

(3) 基地址——上述函数段的起始地址。

## 13.9 映像列表文件控制选项

映像列表文件控制选项与编译列表文件控制选项有些相像。表 13.2 给出映像文件的控制选项。

表 13.2 映像列表文件的控制选项

控制选项	作用
PRINT(映像文件名)	指定映像文件名。缺省为第一目标文件主名加扩展名.M51
PAGELNGTH(n)	映像文件每页行数(0~65 535), 缺省为 68
PAGELINES(m)	映像文件每行字符数(80~135), 缺省为 80
[NO] MAP	输出或禁止各存储空间的映像
[NO]SYMBOLS	输出或禁止符号表
[NO]PUBLICS	输出或禁止公用符号
[NO]LINES	输出或禁止行号
IXREF	产生交叉引用报告
IXREF(NOGENERATED)	产生交叉引用报告, 但不包括编译连接时内部自动生成的带?头的部分

## 13.10 连接/定位命令

下面给出连接/定位命令行的两种格式。

格式:

L51 目标文件名表 [TO 绝对目标文件名][控制选项表] <CR>

L51 @命令文件名 <CR>

其中:

(1) 目标文件名表——用逗号分隔的各目标文件名。目标文件名的组成有: 各种语言经翻译后得到的目标文件(扩展部为.obj), 后跟自建库文件名(如果有的话)。标准库函数的连

接是自动的,不需要在命令行中列出。

(2) 自建库文件名——可带参数以指出选用其中的那些函数模块。自建库文件名格式如下:

自建库文件名 1[(函数模块 1,函数模块 2,...)] [,自建库文件名 2[(函数模块 A,函数模块 B,...)]...

(3) [绝对目标文件名]——用户指定的用于存放生成的绝对目标模块的文件名。它是可给可不给的。不给时用主目标文件的主名作为缺省的绝对目标文件名。目标文件表中的第一个文件名称为主文件名。通常使用缺省名,只在特殊需要时才给绝对目标文件名。

(4) [控制选项表]——可有可无的。没有时,用的是控制缺省的选项和源文件中的控制伪指令行。命令行上可使用的控制选项较多,可分为三类:连接控制选项、定位控制选项和映像列表控制选项。它们分别已在前面的两节中叙述过了。

(5) 命令文件名——命令文件名是把连接/定位命令行上除命令本身(即 L51)之外的全部存放在命令文件中,以便在命令行上用命令文件名实现原命令行上同样的功能。

(6) 命令文件名常起名为 LINK.L51。

(7) L51 命令行要求所有的命令行控制选项都放在一行之中。但是,DOS 的一行限为最多 128 个字符,为此,L51 支持用续行符 & 转到 DOS 的下一行,以继续同一命令行的输入。改用命令文件名之后,就可以按照 DOS 对一行的要求书写,不再使用续行符 &。然而,更大好处是当重复操作时,省去每次敲入冗长的命令行。

[例 13.3]

```
C:\C51> L51 SAMPLE1.OBJ, SAMPLE2.OBJ, SAMPLE3.OBJ, &<CR>
>> UTILITY.LIB IXREF <CR>
```

根据缺省控制,在本目录下生成名为 SAMPLE1 的绝对目标文件和名为 SAMPLE1.M51 的映像列表文件。而映像列表文件中包含有调试用的符号、公用符号和行号。除此之外,根据命令行的控制选项 IXREF 的要求在映像列表文件中加入交叉引用表。

[例 13.4]

```
C:\C51> L51 SAMPLE1.OBJ, SAMPLE2.OBJ, SAMPLE3.OBJ, &<CR>
>> UTILITY.LIB(FPMUL, FPDIV) IXREF(NOGENERATED) <CR>
```

与上不同的是:

- 只将自定义库 UTILITY.LIB 中的 FPMUL 和 FPDIV 模块(浮点乘与除)链入。
- 映像文件的交叉引用表只包含用户定义的符号,而不包含内部生成的符号。

[例 13.5]

```
C:\C51> L51 SAMPLE1.OBJ, SAMPLE2.OBJ, SAMPLE3.OBJ, &<CR> >>
UTILITY.LIB(FPMUL, FPDIV) IXREF(NOGENERATED) &<CR>
>> RAMSIZE(256) PRECEDE(VAR) BIT(40H) DATA(30H) &<CR>
>> IDATA(80H) XDATA(1000H, ? XD? CMODE3(0E000H), &<CR>
>> ? XD? FUNC1? CNODE3) CODE(CODESEG1(4000H), &<CR>
>> CODESEG2(8000H)) STACK(STACK_AERA) <CR>
```

本例除例 13.4 的内容外,加了定位控制选项如下:

·VAR 定位在内部 RAM 的前区空间的最低位置上,BIT 段定位在位地址 40H 处,相当于 28H。

·DATA 段定位于片上 RAM 的 30H 处。

- IDATA 段定位于片上 RAM 的 80H 处。
- 片外 RAM 定位在 1000H 处开始。
- 编译模块 CMODE3 的全局变量所在的段? XD? CMODE3 的基地址定位在片外 RAM 的 0E000H 处。

- 同一编译模块的? XD? FUNC1? CNODE3 段接下去安放。
- 可再定位段 CODESEG1 和 CODESEG2 分别定位于 ROM 空间的 4 000H 和 8 000H 处。
- 堆栈段 STACK\_AERA 放在内部 RAMIDATA 的未用空间。

[例 13.6] 因为命令行太长,宜把命令行的参数部分全部放在命令文件之中。命令文件起名 LINK.L51。这时,将命令行改为:

```
C:\C51> L51 @LINK.L51 <CR>
```

[例 13.7] 调试正确之后应再连接定位一次。这次把所有由于缺省控制而生成的调试信息和定位映像信息全部删掉,只留下干净的执行代码。

```
C:\C51> L51 SAMPLE1.OBJ, SAMPLE2.OBJ, SAMPLE3.OBJ &<CR>
>> UTILITY.LIB(FPMUL, FPDIV) <CR>
```

## 13.11 特殊连接控制选项示例

当编写的程序中引用了函数地址或参数中有函数指针或中断函数中又调了函数时,要仔细检查程序有没有直接或间接递归调用问题,有无会被 L51 误解之处。如有应加带参数的 OVERLAY 控制选项进行引导。

[例 13.8] 本例为单一文件程序,文件名为 ovl1.c。文件中有一个函数 exect( )用函数指针做参数。主函数 main( )中,if 条件语句两次通过 exect( )调不同的函数。这两个不同的函数是 indirectfunc1 和 indirectfunc2。

```
1      /* ovl1.c */
2      #define SWITCH 1
3      ...
4
5
6      bit indirectfunc1( )
7      {
8          1      unsigned char n1, n2;
9          1      return( n1 < n2 );
10         1      }
11
12         1      bit indirectfunc2( )
13         {
14             1      unsigned char a1, a2;
15             1      return((a1 - 0x41) < (a2 - 0x41));
16             1      }
17
18         void exect(bit( * fct)( ))
19         {
20             1      unsigned char i;
```





```

21 1      for(i=0; i<10; i++)
22 1      if( *fct)i += 10 ;
23 1      |
24
25      void main( )
26      {
27 1      bit indirectfunc1( ), indirectfunc2() ;
28 1
29 1      if(SWITCH)
30 1      exect(indirectfunc1) ;
31 1      else
32 1      exect(indirectfunc2) ;
33 1      |
    ...
    ...
    ...

```

OVERLAY MAP OF MODULE: OVL1. (OVL1)

SEGMENT	BIT_GROUP	DATA_GROUP
+ - - - > CALLING SEGMENT	START LENGTH	START LENGTH
? C-C51STARTUP	— —	— —
+ - - - > ? PR? MAIN? OVL1		
? PR? MAIN? OVL1	— —	— —
+ - - - > ? PR? INDIRECTFUNC1? OVL1		
+ - - - > ? PR? EXECT? OVL1		
+ - - - > ? PR? INDIRECTFUNC2? OVL1		
? PR? INDIRECTFUNC1? OVL1	— —	0008H 0002H
? PR? EXECT? OVL1	— —	0008H 0004H
? PR? INDIRECTFUNC2? OVL1	— —	0008H 0002H

由 OVERLAYMAP 表可以看出 L51 分不清 main( ) 对 INDIRECTFUNC1( ) 和 INDIRECTFUNC2( ) 的调用是通过 EXECT( ) 的函数指针间接调用的, 误认为它直接调用了 EXECT( ), INDIRECTFUNC1( ), INDIRECTFUNC2( ) 从而将它们之间的局部变量不正确地覆盖了。为此, 程序员应在命令行上应加特殊的连接控制选项如例 13.9。

#### [例 13.9]

```

C:\C51>L51 OVL1.OBJ OVERLAY(main~(indirectfunc1, indirectfunc2)&<cr>
>>, exect!(indirectfunc1, indirectfunc2))<CR>

```

经再一次连接/定位, 由这次得到的 OVERLAYMAP 表可以看出其间的调用关系和覆盖关系已经摆正。

超星阅读器提示：  
使用本复制品  
请尊重相关知识产权！

OVERLAY MAP OF MODULE: OVL1. (OVL1)

SEGMENT	BIT_GROUP		DATA_GROUP	
	START	LENGTH	START	LENGTH
+ - - - > CALLING SEGMENT				
? C-C51STARTUP	—	—	—	—
+ - - - > ? PR? MAIN? OVL1				
? PR? MAIN? OVL1	—	—	—	—
+ - - - > ? PR? EXECT? OVL1				
? PR? EXECT? OVL1	—	—	0008H	0004H
+ - - - > ? PR? INDIRECTFUNC1? OVL1				
+ - - - > ? PR? INDIRECTFUNC2? OVL2				
? PR? INDIRECTFUNC1? OVL1	—	—	000CH	0002H
? PR? INDIRECTFUNC2? OVL2	—	—	000CH	0002H

[例 13.10] 本例为单一文件程序, 文件名为 ovl2.c。主函数 main( ) 中通过函数的指针数组间接调用函数 func1( ) 和 func2( )。这种间接调用, 只要不发生递归调用, 本来是没有问题的。然而, 问题发生在函数指针数组本身被说明为常量指针数组, 因而, 理所当然地被放在 CODE 空间。又因为函数 func1( ) 和 func2( ) 的内部正好都存在有字符串常量, 字符串常量也是理所当然地应放在 CODE 空间。L51 把调用链中两次访问 CODE 空间的常量区误认为是对函数的递归调用, 从而报错。下面给出 ovl2 模块的部分列表清单。它是没有问题的, 但是报错。随后的例 13.11 给出程序员的特殊引导。

OVL2 模块的部分映像列表如下:

```

1      /* OVL2.C */
...
4      void func1( )
5      {
6      1      unsigned char i ;
7      1      for(i=0; i<10; i++ ) printf("this is function1/n") ;
8      1      |
9
10     void func2( )
11     {
12     1      unsigned char i ;
13     1      for(i=0; i<10; i++ ) printf("this is function2/n") ;
14     1      |
15
16     code void (* functable[ ])( ) = { func1, func2 } ;
17

```



```

18      void    main( )
19      |
20      1        (* funtable[P1 & 0x01])( ) ;
21      1        |

```

...

[例 13.11] 程序员给出的特殊引导如下。

C:\C51>L51 OVL2.OBJ OVERLAY(? CO? OVL2~(func1,func2), &<cr>

>> main ! (func1,func2)) <CR>

L51 不再报错。正确的 OVERLAYMAP 表如下：

OVERLAY MAP OF MODULE: OVL2. (OVL2)

SEGMENT	BIT_GROUP	DATA_GROUP
+ - - - > CALLING SEGMENT	START LENGTH	START LENGTH
? C-C51STARTUP	— —	— —
+ - - - > ? PR? MAIN? OVL2		
? PR? MAIN? OVL2	— —	— —
+ - - - > ? C_LIB_CODE		
+ - - - > ? CO? OVL2		
+ - - - > ? PR? FUNC1? OVL2		
+ - - - > ? PR? FUNC2? OVL2		
? PR? FUNC1? OVL2	— —	0008H 0001H
+ - - - > ? CO? OVL2		
+ - - - > ? PR? PRINTF? PRINT		
? PR? PRINTF? PRINT	— —	0009H 0014H
+ - - - > ? C_LIB_CODE		
+ - - - > ? PR? PUTCHAR? PUTCHAR		
? PR? PUTCHAR? PUTCHAR	— —	001DH 0001H
? PR? FUNC2? OVL2	— —	0008H 0001H
+ - - - > ? CO? OVL2		
+ - - - > ? PR? PRINTF? PRINT		

## 13.12 使用 C51 和 L51 的完整示例

本例是一个完整的多文件程序示例。它体现下列几点：

- 多模块编程原则。
- C51 编译控制选项的使用。
- L51 连接定位控制选项的使用。

超星阅读器提醒：  
使用本复制品  
请尊重相关知识产权！

### 13.12.1 多模块编程

本程序分三个文件编写。

·文件 CSAMPLE1.C 放主函数。要求用户输入两个整型数和加法或减法运算符, 然后向屏幕输出结果。主函数用了两个自编的外部函数和其他库函数。两个自编的外部函数, 一个用于输入一个用于输出。

·文件 CSAMPLE2.C 放自编的输入函数和其他有关的自编函数。

·文件 CSAMPLE3.C 放自编的输出函数。

三个文件的内容在下节的编译列表文件中可以找到。

### 13.12.2 多模块编译

下面分三次编译上述三个文件。编译命令行可以在编译列表的第 5 行找到。

C51 COMPILER V3.0 CSAMPLE1

PAGE 1

DOS C51 COMPILE V3.0, COMPILATION OF MODULE CSAMPLE1

OBJECT MODULE PLACED IN CSAMPLE1.OBJ

COMPILER INVOKED BY C51 CSAMPLE1.C DEBUG

stmt	level	source
1		/* csample1.c C-COMPILER-51 sample program */
2		#include <reg51.h>
3		#include <stdio.h>
4		
5	1	extern int getnumber( );
6	1	extern void output(int );
7		
8		void main( )
9		{
10	1	int number1, number2, result ;
11	1	bit opration ;
12	1	
13	1	SCON = 0x52; /* serial port (2400 baud @12mhz) */
14	1	TMOD = 0x20 ;
15	1	TCON = 0x69 ;
16	1	TH1 = 0xf3 ;
17	1	
18	1	printf(:" \n \n C-COMPILER-51 demonstration program \n \n") ;
19	1	while(1)
20	1	{
21	2	number1 = getnumber( ) ;

```

22      2      number2 = getnumber( ) ;
23      2      printf("input operation: '+' or '-' ?") ;
24      2      operation = (getchar( ) == '+' ) ;
25      2      output(operation ? (number1 + number2) : (number1 - number2) ;
26      2      }
27      1      |

```

C51 COMPILATION COMPLETE. 0 WARNING(S) 0 ERROR(S)

C51 COMPILER V3.0 CSAMPLE2 PAGE 1

DOS C51 COMPILE V3.0, COMPILATION OF MODULE CSAMPLE2

OBJECT MODULE PLACED IN CSAMPLE2.OBJ

COMPILER INVOKED BY C51 CSAMPLE2.C DEBUG

stmt	level	source
1		/* csample2.c C-COMPILER-51 sample program */
2		
3		#include <stdio.h>
4		
5		void getline(char * line )
6		{
7	1	while(( * line++ = getchar( )) != 'n' ) ;
8	1	}
9		
10		int atoi(char * line)
11		{
12	1	bit sign ;
13	1	int number ;
14	1	
15	1	for( ; * line == ' '    * line == 'n'    * line == 't' ; line++ ) ;
16	1	sign = 1 ;
17	1	if( * line == '+'    * line == '-' ) sign = ( * line == '+' ) ;
18	1	for(number = 0 ; * line >= '0' && * line <= '9' ; line++ )
19	1	number = number * 10 + ( * line - '0' ) ;
20	1	return( sign ? number : - number ) ;
21	1	}
22		
23		nusigned int getchar( )
24		{
25	1	char line[40] ;
26	1	
27	1	printf("input number:") ;
28	1	getline(line) ;
29	1	return(atoi(line) ) ;

30 1 |  
31

C51 COMPILATION COMPLETE. 0 WARNING(S) 0 ERROR(S)

C51 COMPILER V3.0 CSAMPLE3

PAGE 1

DOS C51 COMPILE V3.0, COMPILATION OF MODULE CSAMPLE3

OBJECT MODULE PLACED IN CSAMPLE3.OBJ

COMPILER INVOKED BY C51 CSAMPLE3.C DEBUG

stmt	level	source
1		/* csample3.c C-COMPILER-51 sample program */
2		
3		#include <stdio.h>
4		char dummybuf[25];
5		void output(int number)
6		{
7	1	printf(" \ nresult: %d \ n \ n", number);
8	1	{
9		

C51 COMPILATION COMPLETE. 0 WARNING(S) 0 ERROR(S)

### 13.12.3 多模块连接定位

多模块连接定位时的连接命令行可以在映像列表文件的第 4 行上找到。

MCS-51 LINKER/LOCATER L51 V2.0

PAGE 1

MS-DOS MCS-51 LINKER/LOCATER L51 V2.0, INVOKED BY:

L51 CSAMPLE1.OBJ, CSAMPLE2.OBJ, CSAMPLE3.OBJ PRECEDE(? DT? CSAMPLE3) IXREF

MEMORY MODULE: SMALL

INPUT MODULE INCLUDED:

CSAMPLE1.OBJ (CSAMPLE1)

CSAMPLE2.OBJ (CSAMPLE2)

CSAMPLE3.OBJ (CSAMPLE3)

C:\C\C51S.LIB (? C \_ STARTUP)

C:\C\C51S.LIB (? C \_ CLDPTR)

C:\C\C51S.LIB (? C \_ CSTPTR)

C:\C\C51S.LIB (? C \_ IMUL)

C:\C\C51S.LIB (? C \_ PLDHIIDATA)

C:\C\C51S.LIB (PRINT)

C:\C\C51S.LIB (GETCHAR)  
 C:\C\C51S.LIB (? C\_CLDOPTR)  
 C:\C\C51S.LIB (? C\_CCASE)  
 C:\C\C51S.LIB (PUTCHAR)  
 C:\C\C51S.LIB (\_GETKEY)



# LINKER MAP OF MODULE:CSAMPLE1 (CSAMPLE1)

TYPE	BASE	LENGTH	RECOLOCATION	SEGMENT NAME
* * * * * DATA MEMORY * * * * *				
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
DATA	0008H	0019H	UNIT	? DT? CSAMPLE3
DATA	0021H	0001H	BIT_ADDR	? DB? PRINTF? PRINTF
BIT	0022H.0	0000H.1	UNIT	? BI? GETCHAR
BIT	0022H.1	0000H.2	UNIT	"BIT_GROUP"
	0022H.3	0000H.5		* * * GAP * * *
DATA	0023H	0001H	UNIT	? DT? GETCHAR
DATA	0024H	0043H	UNIT	"DATA GROUP"
IDATA	0067H	0001H	UNIT	? STACK
* * * * * CODE MEMORY * * * * *				
CODE	0000H	0003H	ABSOLUTE	
CODE	0003H	0052H	UNIT	? CO? CAMPLE1
CODE	0055H	006AH	UNIT	? PR? MAIN? CSAMPLE1
CODE	00BFH	0010H	UNIT	? CO? CAMPLE2
CODE	00CFH	00ECH	UNIT	? PR? ATOI? CSAMPLE2
CODE	01B8H	002EH	UNIT	? PR? GETNUMBER? CSAMPLE2
CODE	01E9H	0016H	UNIT	? PR? GETLINE? CSAMPLE2
CODE	01FFH	000EH	UNIT	? CO? CAMPLE3
CODE	0200H	0016H	UNIT	? PR? OUTPUT? CSAMPLE3
CODE	0223H	000CH	UNIT	? C_C51STARUP
CODE	022FH	00A8H	UNIT	? C_LIB_CODE
CODE	02D7H	0296H	UNIT	? PR? PRINTF? PRINTF
CODE	056DH	0013H	UNIT	? PR? GETCHAR? GETCHAR
CODE	0580H	0003H	UNIT	? PR? GETCHAR? UNTCHAR
CODE	0583H	0029H	UNIT	? PR? PUTCHAR? PUTCHAR
CODE	05ACH	000AH	UNIT	? PR? _GETKEY? _GETKEY

# OVERLAY MAP OF MODULE: CSAMPLE1 (CSAMPKE1)

SEGMENT	BIT_GROUP	DATA_GROUP
+ - - - > CALLING SEGMENT	START LENGTH	START LENGTH
? C-C51STARUP	—	—
+ - - - > ? PR? MAIN? CSAMPLE1		

? PR? MAIN? CSAMPLE100	22H.1 - -	0000H.1	0024H	0006H
+ - - - >? CO? CSAMPLE1				
+ - - - >? PR? PRINTF? PRINTF				
+ - - - >? PR? GETNUMBER? CSAMPLE2				
+ - - - >? PR? GETCHAR? GETCHAR				
+ - - - >? PR? OUTPUT? CSAMPLE3				
? PR? PRINTF? PRINTF	—	—	0052H	0014H
+ - - - >? C_LIB_CODE				
+ - - - >? PR? PUTCHAR? PUTCHAR				
? PR? PUTCHAR? PUTCHAR	—	—	0066H	0001H
? PR? GETNUMBER? CSAMPLE2	—	—	0066H	0001H
+ - - - >? CO? CAMPLE2				
+ - - - >? PR? PRINTF? PRINTF				
+ - - - >? PR? GETLINE? CSAMPLE2				
+ - - - >? PR? ATOI? CSAMPLE2				
? PR? GETLINE? CSAMPLE2	—	—	0052H	0003H
+ - - - >? PR? PUTCHAR? PUTCHAR				
+ - - - >? C_LIB_CODE				
? PR? GETCHAR? GETCHAR	—	—	—	—
+ - - - >? PR? GETKRY? GETKEY				
+ - - - >? PR? PUTCHAR? PUTCHAR				
? PR? ATOI? CSAMPLE2	0022H.2	0000H.1	0052H	0005H
+ - - - >? C_LIB_CODE				
? PR? OUTPUT? CSAMPLE3	—	—	002AH	0002H
+ - - - >? CO? CAMPLE3				
+ - - - >? PR? PRINTF? PRINTF				

## SYMBOL TABLE OF MODULE: CSAMPLE1 (CSMPLE1)

VALUE	TYPE	NAME
—	—	—
—	MODULE	CSAMOPLE1
C:0005H	PUBLIC	MAIN
—	PROC	MAIN
D:0024H	SYMBOL	NUMBER1



D:0026H	SYMBOL	NUMVER2
D:0028H	SYMBOL	RESULT
D:0022H	SYMBOL	OPERATION
—	ENDPROC	MAIN
C:0055H	# LINE	10
C:0055H	# LINE	14
C:0058H	# LINE	15
C:005BH	# LINE	16
C:005EH	# LINE	17
C:0061H	# LINE	19
C:0070H	# LINE	21
C:0070H	# LINE	22
C:0077H	# LINE	23
C:007EH	# LINE	24
C:0080H	# LINE	25
C:009BH	# LINE	27
C:00BEH	# LINE	29
—	ENDMOD	CSAMOPLE1
—	MODULE	CSAMOPLE2
C:00CFH	PUBLIC	ATOI
C:01B8H	PUBLIC	GETNUMBER
C:01E9H	PUBLIC	GETLINE
—	PROC	ATOI
D:0052H	SYMBOL	LINE
B:0022H.2	SYMBOL	SIGN
D:0055H	SYMBOL	NUMBER
—	ENDPROC	ATOI
C:00CFH	# LINE	10
C:00CFH	# LINE	15
C:0109H	# LINE	18
C:010BH	# LINE	19
C:0142H	# LINE	22
C:0172H	# LINE	23
C:019BH	# LINE	22
C:01A8H	# LINE	25
C:01BAH	# LINE	26
—	PROC	GETNUMBER
D:002AH	SYMBOL	LINE
—	ENDPROC	GETNUMBER
C:01BBH	# LINE	28
C:01BBH	# LINE	31
C:01CAH	# LINE	32

超星阅读器提醒您：  
使用本复制品  
请尊重相关知识产权！



```

C:01D9H      # LINE      33
C:01E8H      # LINE      34
—            PROC        GETLINE
D:0052H      SYMBOL      LINE
—            ENDPROC      GETLINE
C:01E9H      # LINE      6
C:01E9H      # LINE      7
C:01FEH      # LINE      8
—            ENDMOD       CSAMOPLE2

—            MODULE      CSAMOPLE3
D:0008H      PUBLIC      DUMMYBUF
C:020DH      PUBLIC      OUTPUT
—            PROC        OUTPUT
D:002AH      SYMBOL      NUMBER
—            ENDPROC      OUTPUT
C:020DH      # LINE      8
C:020DH      # LINE      9
C:0222H      # LINE      10
—            ENDMOD       CSAMOPLE3

```



## INTER\_MODULE CROSS\_REFERENCE LISTING

NAME \_\_\_\_\_ USAGE MODULES

```

? ATOI? BIT _____ BIT;      CSAMPLE2
? ATOI? BYTE _____ DATA;   CSAMPLE2
? C_CCASE _____ CODE;      ? C_CCASEPRINTF
? CLDOPTR _____ CODE;      ? CLDOPTRPRINTF
? CLDPTR _____ CODE;      ? CLDPTRPRINTFCSAMPLE2
? C_CSTPTR _____ CODE;      ? C_CSTPTR_PRINTF_CSAMPLE2
? C_IMUL _____ CODE;      ? C_IMUL_PRINTFCSAMPLE2
? C_PLDIIDATA _____ CODE;   ? C_PLDIIDATAPRINTFCSAMPLE2
? C_STARUP _____ CODE;      ? C_STARUPCSAMPLE1
? GETLINE? BYTE _____ DATA; CSAMPLE2
? GETNUMBER? BYTE _____ DATA; CSAMPLE2
? MAIN? BIT _____ BIT;      CSAMPLE1
? MAIN? BYTE _____ DATA;   CSAMPLE1
? OUTPUT? BYTE _____ DATA; CSAMPLE3CSAMPLE1
? PRINTF? BYTE _____ DATA; PRINTFCSAMPLE1CSAMPLE2CSAMPLE3
? PUTCHAR? BYTE _____ DATA; PUTCHARGETCHARPRINTF
? SPRINTF? BYTE _____ DATA; PRINTF

```

? UNGETCHAR?	BYTE	DATA;	GETCHAR
ATOI		CODE;	CSAMPLE2
DUMMY_BUF		DATA;	CSAMPLE3
GETCHAR		CODE;	GETCHARCSAMPLE1CSAMPLE2
GETLINE		CODE;	CSAMPLE2
GETNUMBER		CODE;	CSAMPLE2CSAMPLE1
MAIN		CODE;	CSAMPLE1? C_STARTUP
OUTPUT		CODE;	CSAMPLE3CSAMPLE1
PRINTF		CODE;	PRINTFCSAMPLE1CSAMPLE2CSAMPLE3
PUTCHAR		CODE;	PUTCHARGETCHARPRINTF
SPRINTF		CODE;	PRINTF
UNGETCHAR		CODE;	GETCHAR
_GETKEY		CODE;	_GETKEYGETCHAR

值得特别提出的是连接命令行中用了 PRECEDE(? DT? CSAMPLE3)定位选项,堆栈可用字节增加了不少。否则,定位空间映像如下。

TYPE	BASE	LENGTH	RECOLOCATION	SEGMENT NAME
* * * * * DATA MEMORY * * * * *				
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
DATA	0008H	0001H	UNIT? DT	? GETCHAR
		0009H	0017H	* * * GAP * * *
DATA	0020H	0001H	BIT_ADDR	? DB? PRINTF? PRINTF
BIT	0021H.0	0000H.1	UNIT	? BI? GETCHAR
BIT	0021H.1	0000H.2	UNIT	"BIT_GROUP"
		0021H.3	0000H.5	* * * GAP * * *
DATA	0022H	0019H	UNIT	? DT? CSAMPLE3
DATA	003BH	0043H	UNIT	"DATA GROUP"
IDATA	007EH	0001H	UNIT	? STACK

# 第三部分

超星数字图书馆  
使用本馆制品  
请尊重相关知识产权!

## XAC

### (80C51XA 用 16 位嵌入式 C 语言)

## 第十四章 XAC 说明

### 14.1 XAC 变量说明

下面把与 C 语言基础部分变量说明基本相同的部分放在 XAC 一般变量说明中, 把 XAC 特有的部分分别放在 XAC 绝对变量说明和 XAC 位变量说明里。

#### 14.1.1 XAC 一般变量说明

格式如下:

[存储类说明符①] 类型说明符① [修饰符⑤] 标识符② [= 初值③][, 变量标识符 [= 初值]③]…;

①	①	⑤	②	③
<div>auto register static extern typedef</div>	<div>char unsigned char int unsigned long unsigned long float double long double void typedef 别名 bit:</div>	<div>const volatile near far code persistent</div>	<div>变量名 * 指针名</div>	<div>= 变量初值 = &amp; 变量 @ 绝对地址</div>

其中:

(1) 类型说明符①和标识符②两部分是必不可少的。

(2) 定义性说明与引用性说明。

定义性说明为对象分配内存。标识符②为对象起名字, 类型说明符①为对象分配内存的大小, 这两项是必须有的。下面各项是可有可无的: [存储类说明符③]指定存储空间的属性; [修饰符④]对对象的属性做进一步的修饰说明; [=初值⑤]部分为对象赋初值; [, 变量标识符 [=初值]⑥]用逗号分隔符使一个语句为多个同类变量做说明。定义性说明除不能使用 `extern` 和 `typedef` 两个存储类选项外, 其余都可以根据需要选用。

引用性说明是不为对象分配内存的。它用于单文件程序的超前引用或多文件程序的非定义文件中, 告知编译器在引用性说明中所列出的变量仅是引用, 无须分配内存。正因为如此, 引用性说明除说明中必须存在的类型说明符①和标识符②两部分之外, 只须加上 `extern` 存储类选项, 以说明曾经在外部做过定义性说明, 其他选项一概不再需要。

(3) 变量在内存中的存放方法。

变量在 51XA 内存中的存放方法没有沿袭 8051 的“高字节放低地址”的规律, 而是改用“低字节放低地址”。具体安排见图 14.1。浮点数在 51XA 内存中的安排见图 14.2。

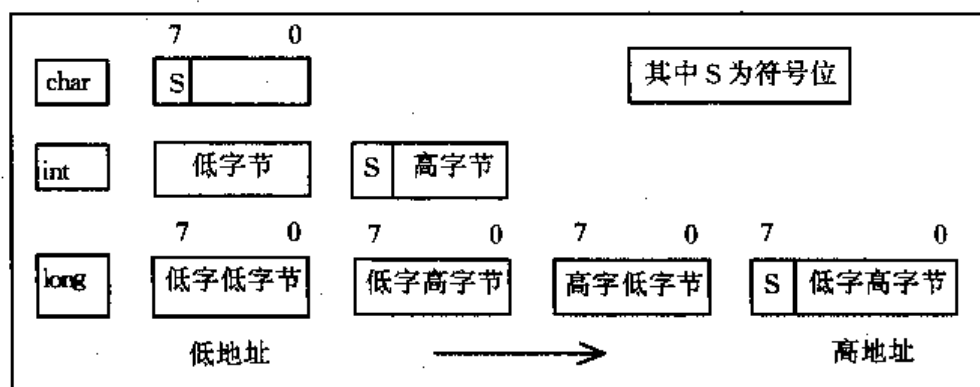


图 14.1 变量在 51XA 内存中的存放方法

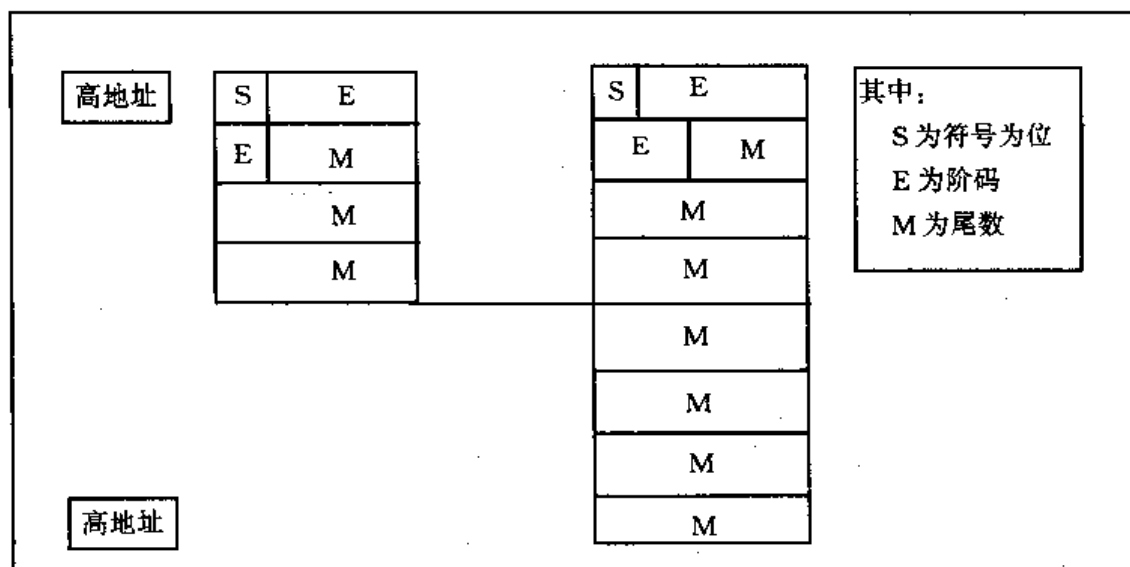


图 14.2 浮点数在 51XA 内存中的安排

int, unsigned, long, unsigned long, float, double 都有向偶地址对齐的要求。

(4) 习惯上[修饰符⑤]在 XAC 中放在类型说明符①之前。

(5) 格式中有一些带下划线的项是 XAC 对 ANSI C 的扩展部分,是基础部分所没有的。修饰部分占有两项:code, persistent。下面即将介绍。还有两项@绝对地址、bit,将分别在绝对变量和位变量两节中介绍。

(6) [修饰符⑤]部分的 code 修饰符:一般微控制器的片上 RAM 有限,而常量(包括字符串)总希望放在 ROM 中去,特别是对于 small 和 medium 模式。使用 code 修饰符后它们即被放入 ROM,对它们的访问仅限于读,而且,必须使用 MOVC A, MOVC @A+DPTR 指令。对于经 static 修饰的带初值的静态变量,也可以用 code 修饰将它放到 ROM 中去。这时,只对 medium 和 large 模式才有意义,因为 small 模式本来就是放到 ROM 中的。

[例 14.1] code int count=0x3333;

```
static code int count=0x3333;
```

[例 14.2] void code \_puts(code char \*codeptr)

```
{ char ch;
  while (ch = *codeptr++)
    putchar(ch);
  putchar('\n');
}

code char hello[] = "hello world!";

void main( )
{
  code _puts(hello);
}
```

由本例可以看出,把字符串放到 ROM 空间,访问时使用的是 code 指针。所以,不能使用 puts(),更不能使用 printf()。另外,code 指针不能用 far 修饰,也不能用于缺省指针的地方。

(7) [修饰符⑤]部分的 persistent 修饰符:对于 static 变量有时希望在热启动、复位和电源意外失电又复电时保持其历史值,以便正确地接续工作。但是,如果不加特别处理,上述情况下,对于 static 变量的处理将和冷启动一样,即带初值的赋初值,不带初值的赋零值。为此,有在上述情况下正确地接续工作要求的,应集中放到非易失存储器中去,如 eeprom 或 flash 存储器。这时,相应地应给 static 变量加上 persistent 修饰符。XAC 的库函数还特别增加了核对和初始化 persistent 变量的专用函数可以利用。

persistent 和 far 同用,可以把 persistent 变量集中到 farnvram 程序段中去,然后可随意安排非易失存储器到 51XA 的任意地址空间,特别是高位地址处有非易失存储器者。

(8) [修饰符⑤]部分的 near 修饰符:虽然在 ANSI C 中已有,但具体到 51XA 意义有所不同。在 XAC 中经 near 修饰过的变量,首先放在片上 RAM 中;其次,对于不同的存储模式处理上不同。区别如下:

small 模式:因为外部变量和 static 变量都是放在片上 RAM,所以,可以不加 near 修饰符。访问方式使用直接地址。

medium 和 large 模式:用 near 修饰的非初值 static 变量放在片上 RAM 的 psect rbss 程序子段中。访问使用寄存器间址。用 near 修饰的带初值 static 变量放在片上 RAM 的 psect rdata 程序子段中。编译时放进 ROM 空间,运行时由启动程序复制到片上 RAM,访问使用寄存器间址。

#### [例 14.3]

```
static near int result;
static near int salary = 200;
```

用 small 模式编译时, result 和 salary 都放在片上 RAM, 并使用直接地址访问。用 medium 和 large 模式编译时, result 放在片上 RAM; salary 编译时放进 ROM 空间, 运行时复制到片上 RAM。访问都使用寄存器间址。

(9) [修饰符●]部分的 far 修饰符:虽然在 ANSI C 中已有,但具体到 51XA 意义有所不同。static 变量用 far 做修饰,变量将被放在当前体外的 RAM 体内,并用段寄存器 ES 指定 RAM 体。

#### [例 14.4]

```
static far int fint;
far int f_int;
```

用 medium 和 large 模式编译,它们都放在 ES 指定的 RAM 体内。

#### [例 14.5]

```
far char *fptr;
```

它定义一个远指针。用远指针可以访问当前和其他 RAM 体内的变量。访问当前 RAM 体内的变量时,用 DS 的内容为指针的高 8 位。访问非当前 RAM 体内的变量时,用 ES 的内容为指针的高 16 位,即这时的指针为 32 位。

#### (10) 关于 XAC 的一般变量指针

·near/far 指针——near 指针是 16 位指针,在当前数据存储器(64 KB)内寻址。far 指针是 32 位指针,在当前数据存储器之内或当前数据存储器之外的其他存储器(64 KB)内寻址。在当前存储器内寻址,段址用寄存器 DS。在当前存储器之外寻址,段址用寄存器 ES。

·code 指针——在程序空间寻址。可用 near/far 修饰,成为 16/32 位指针。

·指针可用运算符:

```

+                限于指针 + int 量
-                限于指针 - int 量
++  --
=
==  !  =
<<  >>  =
```

·指针修饰符——在指针定义性说明中的修饰符所修饰的是所指的对象,如果要修饰的是指针本身,修饰符应加在 \* 号与指针名之间。简化格式如下:

[存储类说明符] 类型说明符 [指针修饰符] \* [修饰符] 指针名;

#### [例 14.6]

```
const char * far ptr;
```

定义指向常量的指针, 指针本身放在当前体之外的 RAM 空间。



### 14.1.2 绝对变量与 SFR

XAC 可以把全局变量和静态变量定位在绝对地址上。这时, 定义说明一定要加初值部分。初值部分用: @绝对地址。绝对地址用 0x 开头的十六进制数。绝对变量定义说明常用简化格式如下:

[FAR] 类型说明符号 标识符 @绝对地址;

其中:

绝对地址用 0x 开头的十六进制数。绝对地址超过 64 KB 必须加修饰符 FAR。

[例 14.7] int zip @0x40;

[例 14.8] far int value @0x10000

绝对变量的另一个用法是将特殊功能寄存器的预定义名赋予相应的 SFR 地址, 以便使用预定义名存取 SFR。定义 SFR 常用简化格式如下:

[static] [volatile] 类型说明符号 预定义 SFR 名 @对应的 SFR 绝对地址;

[例 14.9] static unsigned char s0con @0x420;

static unsigned char s0buf @0x460;

### 14.1.3 位变量与可位寻址 SFR

51XA 片上 RAM 0x20~0x3F 的 16 字节地址空间是可以按位寻址的。它们按位排列的地址空间为 0x100~0x1FF, 共 256 位。这个位空间可供用户定义位变量。下面给出定义位变量常用的简化格式:

[static] bit 位标识符 [, 位标识符]...;

[例 14.10] static bit init\_flg;

51XA 片上 RAM 0x400~0x43F 的 64 字节地址空间是可按位寻址的特殊功能寄存器空间。它们按位排列的地址空间为 0x200~0x3FF, 共 512 位。下面是定义可位寻址的 SFR (特殊功能寄存器) 的常用绝对位变量的简化说明格式:

[static] 类型说明符号 预定义 SFR 名\_0~7 @对应位的绝对位地址;

[例 14.11] 为 P0 口的各位 P0\_0~7 赋对应位的绝对位地址。

已知 P0 口的地址为 0x432, 其最位的绝对位地址是:

$$0x200 + (0x432 - 0x400) * 8 = 0x390$$

故有:

```
static bit p0_0 @0x390;
static bit p0_1 @0x391;
static bit p0_2 @0x392;
static bit p0_3 @0x393;
static bit p0_4 @0x394;
static bit p0_5 @0x395;
static bit p0_6 @0x396;
static bit p0_7 @0x397;
```



## 14.2 XAC 数组说明

格式如下:

[存储类说明符①] 类型说明符② [修饰符③] 标识符④ [= 初值⑤][, 标识符④] [= 初值]⑤]...;

①	②	③	④	⑤
auto register static extern typedef	char unsigned char int unsigned long unsigned long float double long double void typedef 别名	const volatile near far cod persistent	数组名[常量表达式] (* 数组指针)[ ] * 指针数组名[常量表达式] * * 二重指针 二维数组名[常量表达式][常量表达式] * 二维数指针组名[常量表达式][常量表达式] * * 二重指针	[= {数组元素初值表}] [= 数组名] [= {指针数组元素初值表}] [= 指针数组名] [= {{行数数组元素表}, ...}] [= {{行指针元素表}, ...}] [= 二维数指针组名] <u>@绝对地址</u>

其中:

(1) 带下划线的部分是基础部分的数组说明中所没有的, 是 XAC 关于 ANSI C 的扩展部分。关于它们的意义在 14.1 节中已经阐明。值得一提的是: XAC 的数组可以直接定位到绝对地址上。

(2) 不带下划线的部分是与基础部分的数组说明相同的部分。值得一提的是: 它们可以看做是关于数组说明的总结, 特别是④和⑤两部分。

(3) 不能有 bit 类型的数组。

(4) 在 XAC 中, 习惯上将修饰符放在类型说明符之前。

## 14.3 XAC 结构说明

XAC 的结构部分与基础部分相同, 这里不再赘述。

## 14.4 XAC 联合说明

XAC 的联合部分与基础部分相同, 这里不再赘述。

## 14.5 XAC 函数说明

本节把与基础部分函数说明基本相同的部分放在 XAC 一般函数说明中, 把 XAC 特有的部分分别放在中断向量表添写和中断接管两节中介绍。

### 14.5.1 XAC 一般函数说明

函数有定义性说明和原型说明。

#### 1. 定义性说明格式

[存储类说明符]⑤ 类型说明符① [修饰符]⑥ 标识符② (参数表③) {函数体}④

⑤	①	⑥	②
[static]	char	interrupt	函数名
	unsigned char	<u>banked</u>	* 函数名
	int	near	( * 函数名 )
	unsigned	far	* ( * 函数名 )
	long		
	unsigned long		
	float		
	double		
	long double		
	struct		
	union		
	void		
	<u>bit</u>		

#### 2. 原型说明格式

extern 类型说明符① [修饰符]③ 标识符② (参数表④);

其中:

(1) 带下划线的部分是基础部分的函数说明中所没有的,是 XAC 关于 ANSI C 的扩展部分。带下划线的部分有修饰符 banked 和类型说明符 bit。banked 是指选用寄存器组。bit 是指函数返回位类型。

(2) 不带下划线的部分是与 C 语言基础部分的函数说明相同的部分。

(3) 在 XAC 中,习惯上将修饰符放在类型说明符之前。

(4) 关于函数原型说明格式——除非某一函数在同一文件之内且在函数定义之后被调用者外,凡在同文件之内超前调用者或在它文件中欲调用此函数者,在调用前必须先做函数的原型说明,然后才能调用。被调函数的原型说明应放在调用函数之外,习惯上放在文件的前部。原型说明格式的特点是:定义说明中的①,②,③三部分必须存在,并在最前面加 extern,在最后加终结符“;”。

### 14.5.2 XAC banked 中断函数说明

给函数加上修饰符 interrupt 形成基础部分所说的一般中断函数。具体到 XAC,因为微控制器 51XA 的通用寄存器是分组的结构,所以规定一般中断函数使用 0 组的 R0~R7。欲在中断函数中使用寄存器组切换方式工作的应再加修饰符 banked。两种方式影响所加的中断前缀段和中断后缀段。一般中断函数将 0 组的 R0~R7 进栈出栈,banked 中断函数仅仅在中断前缀段中,将 PSW 压栈后再修改 PSW 中的寄存器组号即可。但在 XAC 中,预先修改过的 PSW

是提前放在中断向量表中的,所以,这个操作是自动进行的。banked中断函数效率较高。但是,程序员应当保证不会发生寄存器混乱。例如,中断切入的寄存器组正好与正在工作的寄存器组相重。如果没有把握,最好使用一般中断函数。

[例 14.12] 编写串行口的接收中断函数。

```
char rxbuf[16];
volatile char head, tail;

interrupt void serial-rx-int(void)
{
    rxbuf[head++] = sbuf;
    if((head % sizeof(rxbuf)) == tail)
        tail = ++tail % sizeof(rxbuf);
    ri = 0;
}
```



### 14.5.3 中断向量表(ROM 向量表)的添写

XAC 要求用户添写中断向量表,但是,并不要求用户写程序,而是仅仅调用参数宏。一般所说的中断向量表是 ROM 向量表。它是位于 ROM 空间最低端的以中断向量为索引的一张表,其表项一定包含有对应中断函数的入口地址。51XA 的中断向量表的表项占 4 个字节。前两个字节放中断函数的入口地址,后两个字节放 PSW。

XAX 为方便用户填写中断向量表,定义了有关的宏定义和参数宏。它们放在 intrpt.h 头文件中。表 14.1、表 14.2、表 14.3 分别列出与中断有关的参数宏、中断向量值宏定义、PSW 宏定义。

表 14.1 与中断有关的参数宏

参数宏	解 释
ROM_VECTOR(中断向量值,中断函数的入口地址,PSW)	ROM 向量表表项初始化
RAM_VECTOR(中断向量值,中断函数的入口地址,PSW)	RAM 向量表表项初始化
CHAGE_VECTOR(中断向量值,中断函数的入口地址)	改变 RAM 向量表中中断函数的入口地址
READ_RAM_VECTOR( )	获取 RAM 向量表中中断函数的入口地址

表 14.2 中断向量值宏定义

中断向量值宏定义	中断向量值	对应的中断
IV_EX0	0x80	外部中断 0
IV_T0	0x84	定时中断 0
IV_EX1	0x88	外部中断 1
IV_T1	0x8C	定时中断 1
IV_T2	0x90	定时中断 2
IV_RX0	0xA0	串行接收中断 0
IV_TX0	0XA4	串行发送中断 0
IV_RX1	0XA8	串行接收中断 1
IV_TX1	0xAC	串行发送中断 1
IV_SWI1	0x100	软件中断 1

续表 14.2

中断向量值宏定义	中断向量值	对应的中断
IV_SWI2	0x104	软件中断 2
IV_SWI3	0x108	软件中断 3
IV_SWI4	0x10C	软件中断 4
IV_SWI5	0x110	软件中断 5
IV_SWI6	0x114	软件中断 6
IV_SWI7	0x118	软件中断 7

表 14.3 PSW 宏定义

PSW 宏定义	解 释
IV_PSW	标准 PSW 设置(系统模式, 执行优先级 15, 寄存器组 0)
IV_SYSTEM	系统模式置 1
IV_PRIO0	执行优先级 0(最底)
IV_PRIO1	执行优先级 1
IV_PRIO2	执行优先级 2
IV_PRIO3	执行优先级 3
IV_PRIO4	执行优先级 4
IV_PRIO5	执行优先级 5
IV_PRIO6	执行优先级 6
IV_PRIO7	执行优先级 7
IV_PRIO8	执行优先级 8
IV_PRIO9	执行优先级 9
IV_PRIO10	执行优先级 10
IV_PRIO11	执行优先级 11
IV_PRIO12	执行优先级 12
IV_PRIO13	执行优先级 13
IV_PRIO14	执行优先级 14
IV_PRIO15	执行优先级 15(最高)
IV_BANK0	寄存器组 0
IV_BANK1	寄存器组 1
IV_BANK2	寄存器组 2
IV_BANK3	寄存器组 3

表 14.1 中有 ROM 向量表表项初始化用的参数宏, 具体格式如下:

ROM\_VECTOR(中断向量值, 中断函数的入口地址, PSW)

其中:

(1) 中断向量值是每个中断对应的中断向量表索引值。表 1.2 是各个中断所对应的向量值的宏定义, 供填写参数宏时使用。

(2) PSW 对于切换式(banked)中断函数应添写寄存器切换后的 PSW, 对于一般中断函数

应添写标准的 PSW。PSW 的高字节要由如下三部分组成：

7	6	5	4	3	0
系统模式		寄存器组编码		中断屏蔽位编码	

它们的宏定义已在表 14.3 中给出。

[例 12.13] 为串行口中断添写中断向量表,使用一般中断函数。已知:

```
interrupt void serial_rx_int(void) { ; }
```

有:

```
ROM_VECTOR(IV_RX0, serial_rx_int, IV_PSW);
```

[例 14.14] 将例 14.13 改为切换式中断函数,切换到寄存器组 2。已知:

```
interrupt banked void serial_rx_int(void) { ; }
```

有:

```
ROM_VECTOR(IV_RX0, serial_rx_int, IV_SYSTEM + IV_PRIO15 + IV_BANK2);
```

应该指出:① 添写 ROM 中断向量表的宏调用语句,在添写完 ROM 中断向量表之后即被废除,并不进入可执行代码。所以,该宏调用语句可以放在程序的任意地方。② 参数宏的宏体是用嵌入式汇编指令写的,故十六进制数可以用 C 风格,也可以用汇编风格(如 0A0H)。

#### 14.5.4 中断接管与 RAM 向量表

HI-TECK 的 XAC 支持在程序执行中动态地修改中断向量表,使其指向另外的中断函数。由于 ROM 中断向量表在运行其间不能修改,所以,需要在片内 RAM 空间(rdata 子段)再建一个 RAM 向量表。凡是欲被接管的中断都要在 RAM 向量表中立项。应把 RAM 向量表的向量值填入 ROM 中断向量表的表项, RAM 向量表的表项填入 PUSH 和 RET 指令, PUSH 指令的操作数应放新中断函数的入口地址。具体做法仍是调用参数宏。与 RAM 向量表有关的参数宏有 RAM-VECTOR(中断向量值,中断函数的入口地址,PSW)、CHAGE-VECTOR(中断向量值,中断函数的入口地址)、READ-VECTOR( ),用法与 ROM-VECTOR(中断向量值,中断函数的入口地址,PSW)相同。

[例 14.15]

```
volatile unsigned char wait_flag;
interrupt void waiting_int(void)
{
    ++ waiting_flag;
    RI=0;
}

void serial_waiting_exchange(void)
{
    interrupt voids (*oldhandler)(void);
    static bit EA @0x337;
    EA=0;
    oldhandler=READ_RAM_VECTOR(IV_RX0);
    wait_flag=0;
    CHANGE_VECTOR(IV_RX0,waiting_int,IV_PSW);
}
```

```
while(wait_flag == 0) ;  
EA = 0;  
CHANGE_VECTOR(IV_RX0, oldhandler, IV_PSW);  
EA = 1;
```



注意: READ-RAM-VECTOR( ) 必须在初始化过 ROM-VECTOR( ) 和 RAM-VECTOR( ) 之后才能使用。不然, 取得的结果将是无用的。



## 第十五章 XAC 编译器内部管理规范和约定

使用本资料前  
请尊重相关知识版权

### 15.1 XAC 标准程序子段(psect)

XAC 编译器编译时把程序和数据分别放在不同的标准程序子段(psect)中,连接时把子段归并为段(segment)。下面介绍编译器编译时生成的各种标准程序子段(psect)。

**vector psect:** 包括 reset 向量和中断向量表,连接时 reset 向量在前,中断向量表在后。定位在 ROM 空间的零地址开始处。

**text psect:** C 编译器产生的可执行代码和汇编器产生的可执行代码都放在本子段内。本子段定位在 ROM 空间的零体内。对于 small, medium 模式可执行代码都放在本子段内。对于 large 模式的 text psect 子段也是定位在 ROM 空间的零体内的,其他放在 ltext psect 子段的可执行代码定位在 ROM 空间的非零体内。不管什么模式,中断函数都是被放在本子段定位在 ROM 零体之内的。

**code psect:** 说明为 code 类型的静态初值常量放在本子段内。本子段连接时放在 text psect 之后。读出时用 movc 指令。如:

```
code int max=10;
```

**const psect:** 用存储类 const 说明的初值常量放在本子段内。如:

```
const char masks[] = {1,2,4,8,16,32,64,128};
```

**strings psect:** 匿名字符串放在本子段内。匿名字符串直接用于函数的调用中。如:

```
printf("hello world!");
```

**rdta psect:** 经 near 修饰的静态初值变量放在本子段内。静态变量的初值有一份副本保存在 ROM 中,启动时在调 main() 之前将初值复制到 RAM 中。本子段不管什么存储模式其表现一样。如:

```
static near int size=256;
```

**data psect:** 所有不属于 code, const 和 near 修饰过的静态初值变量都放在本子段内。对于 small 存储模式,本子段是被连接到 ROM 空间。因此,不能再改变,只能用 MOVC 指令读出。对于 medium, large 存储模式,本子段是被连接到 RAM 空间的。静态变量的初值有一份副本保存在 ROM 中,启动时在调 main() 之前将初值复制到 RAM 中。

**rbss psect:** 无初值的静态和外部变量, 凡经 near 修饰过的都放在本子段内。本子段被连接到片内 RAM 地址小于 0x400 的空间。启动时在调 main() 之前将本子段内各变量清零,用直接地址访问。

**bss psect:** 无初值的静态和外部变量都放在本子段内。本子段被连接定位到片外 RAM 空间。启动时在调 main() 之前将本子段内各变量清零。

**rbit psect:** 所有的位变量都放在本子段内。绝对位变量因已经定位,所以不包括在内。



位空间地址 0x00~0xFF 是通用寄存器组的各位。0x100~0x1FF 放位变量。0x200~0x3FF 是位寻址的 SFR。如：

static bit flag;

flag 放在位变量区 0x100~0x1FF。

nvrasm psect: 经 persistent 修饰过的变量都放在本子段内。它们在定位时,可由命令行上指定定位到绝对地址上。也可以不指定定位绝对地址,作为缺省被连接到 bss psect 子段之后加以定位。它们在启动时不会被清零或修改。

stack psect: 指定在 RAM 存储器的顶部。不包含任何数据时,其子段长度便是零。heap psect 它的子段长度是零时,指出已用 RAM 的最高位。运行中动态分配存储器时,子段长度由此向上增长,正好与堆栈的增长相对(堆栈向下增长)。

farbss psect: 经 far 修饰过的变量都放在本子段内。它总是被定位在非零体的零地址开始处。

frnvrasm psect: 经 far persistent 说明过的变量都放在本子段内。它可被定位在任何物理 RAM 空间。

## 15.2 XAC 有关寄存器的约定

尽管 51XA 是通用寄存器联合成组的结构,但是,在 XAC 中约定:①无特殊声明时,都假定用的是通用寄存器组 0。②其中的 R0, R1, R2, R3 用于片内 RAM 的间址寄存器和临时变量的存放。③函数前三个参数的传送使用 R3, R2, R1, 返回也使用 R3, R2, R1, 同时也用于存放函数的临时变量;函数中的寄存器变量使用 R4, R5, R1。④编译时进行全局性优化, R3, R2, R1 用于存放参数和的临时变量。

上述约定,对于被 C 调用的汇编程序也应同样遵守。

## 15.3 XAC 有关参数传送和函数返回的约定

函数的前三个参数只要不大于 16 位的都要按顺序用 R3, R2, R1 传送。其中 8 位的实际用的是 R3L, R2L, R1L 传送的。前三个参数有大于 16 位的或参数超过的均放在堆栈内传送。改在堆栈内传送的前三个参数有未用的寄存器应空着不用。

对于有变参数的函数,对应与形参表中“...”号前的一个参数就应改在堆栈内传送。它作为支撑点依次传送后继的多个参数。在堆栈内传送参数,一律以 16 位为单位。

凡是使用堆栈传送过参数的函数,调用函数应负责于函数返回后清理堆栈排除所有的参数。

关于函数返回值,32 位以下的由寄存器返回,根据长度依次由 R0, R1, R2, R3 传送。具体地说,8 位用 R0L 返回,16 位用 R0, 32 位用 R0 和 R1, 64 位的 double 用 R0, R1, R2, R3, 最低有效位在 R0 中。

对于结构和联合,返回的方式因结构和联合的长度而异。小于 64 位的用寄存器返回,规律同上。再大的结构和联合改用指针返回,这时,调用函数应负责将指针放在 R0 中。被调函

数应注意保存这个指针,在被调函数中可以使用这个指针,不过应该保证返回时 R0 中应是原来的指针。

## 15.4 XAC 关于函数的签字

XAC 编译器为每个新定义的函数生成一个标识码,俗称签字。函数的签字是考虑了返回类型、参数个数和类型以及其他属性计算出来的 16 位数值。这个数值附在函数的定义代码和调用代码中。在连接时连接器为它们进行匹配审核。不匹配时告警。

各函数的签字是不公开的。需要的话,可以照函数的原型写一个空函数,单独用产生汇编码的选项进行编译。格式如下:

```
XAC -S X.C
```

其中:

X.C 是含欲求空函数的文件名。-S 是产生汇编码的选项。

在产生汇编码程序中有包括下列伪指令语句:

```
SIGNAT _函数名 签字
```

混合编程时,由 C 所调的汇编子程序应附有签字。为取得这个签字,应写出这个子程序的 C 风格空函数,并按上述格式编译之。

[例 15.1] 写出汇编子程序的 C 风格空函数。

```
/* X.C */
char widget(int arg1, int arg2) { ; }
```

编译之:

```
XAC -S X.C
```

在产生的 X.OBJ 文件中有:

```
SIGNAT _WIDGET, 8249
```

## 15.5 XAC 有关存储器的约定

51XA 硬件提供了两个堆栈:系统堆栈和用户堆栈。系统堆栈指针为 16 位,用户堆栈指针为 24 位。系统堆栈只工作于数据存储空间的第 0 段(64 KB)。高位地址线被强制到 0。用户堆栈可工作于数据存储空间的任意一段,段号由段寄存器提供。由于用户堆栈的使用有许多限制,所以,对于典型的嵌入式应用,XAC 约定用户堆栈所用的 DS 永远置为 0。用户堆栈被限制在 0 段,不用 FAR 修饰的变量,即一般变量都应在 0 段。这就是说,外部存储空间都应映射到 0 段。但是映射有许多困难。对于超过 64 KB 的变量和指针改用 FAR 进行修饰,用 ES 选段存取。使用 XAC,遇 DS 非 0 时将报错。

## 15.6 XAC 的存储模式

XAC 支持三种存储模式,即 small, medium, large 模式。

## 1. 小模式(small)

本模式下,程序空间最大 64 KB,数据空间最大 1 KB,全部为片上 RAM。数据用直接寻址方式。为节省片上 RAM,字符串常量和初值静态变量改放在 ROM 中,用 MOVC 指令读出,运行中不可改变。本模式所用启动程序模块和库程序模块见表 15.1。它们是编译和连接时自动选定的,用户无须干预。

表 15.1 小模式(small)库

RTXA - S.OBJ	小模式运行时启动程序
XA - SC.LIB	小模式标准库, 32 位 double
XA - SL.LIB	小模式 printf 库, long
XA - SF.LIB	小模式 printf 库, long 和 float
XA - DSC.LIB	小模式标准库, 64 位 double
XA - DSL.LIB	小模式 printf 库, long 和 64 位 double
XA - DSF.LIB	小模式 printf 库, float 和 64 位 double

## 2. 中模式(medium)

本模式下,程序空间最大 64 KB,数据空间最大 64 KB。函数指针和数据指针均为 16 位。数据用间址或变址方式寻址。超范围的数据可加 FAR 修饰。片上 RAM(地址 0x400 以下)的数据需用 near 修饰,并用直接方式寻址。本模式所用启动程序模块和库程序模块见表 15.2。它们是编译和连接时自动选定的,用户无须干预。

表 15.2 中模式(medium)库

RTXA - M.OBJ	中模式运行时启动程序
XA - MC.LIB	中模式标准库, 32 位 double
XA - ML.LIB	中模式 printf 库, long
XA - MF.LIB	中模式 printf 库, long 和 float
XA - DMC.LIB	中模式标准库, 64 位 double
XA - DML.LIB	中模式 printf 库, long 和 64 位 double
XA - DMF.LIB	中模式 printf 库, float 和 64 位 double

## 3. 大模式(large)

本模式下,程序空间最大 16 MB(具体数与 51XA 芯片型号有关),使用 51XA 的远调用和远返回。大模式也叫分体模式,由 ROM 的 0 体开始按顺序一个又一个地自动使用。数据空间与 medium 模式一样。本模式所用启动程序模块和库程序模块见表 15.3。它们是编译和连接时自动选定的,用户无须干预。

表 15.3 大模式(large)库

RTXA - L.OBJ	大模式运行时启动程序
XA - LC.LIB	大模式标准库, 32 位 double
XA - LL.LIB	大模式 printf 库, long
XA - LF.LIB	大模式 printf 库, long 和 float
XA - DLC.LIB	大模式标准库, 64 位 double
XA - DLL.LIB	大模式 printf 库, long 和 64 位 double
XA - DLF.LIB	大模式 printf 库, float 和 64 位 double



## 15.7 XAC 关于运行时启动模块的规定

XAC 提供针对 51XA 微控制器各种模式下的运行启动模块。启动模块执行位于 vector psect 中的 reset 向量。将 rbss, bss, rbit 子段清 0, 将 rdata, data 子段复制到 RAM 中等。各种模式下的运行启动模块名已分别列于表 15.1, 表 15.2, 表 15.3 中。

## 15.8 XAC 上电子程序

有些硬件配置要求在 reset 后极短的几个机器周期内进行初始化。51XA 上的集成监督定时器就是一个例子。如果想禁止它, 就得用程序关掉。习惯上, 上电子程序是由用户用汇编指令写成的。所以, 在 reset 向量处安置挂钩(HOOK), 先经挂钩绕到上电子程序, 上电子程序结束时跳到运行启动模块。

下面是用嵌入汇编指令写成的内容为开放集成监督定时器的上电子程序(关于嵌入汇编指令见后)。

```
# asm
    psect text, globol, reloc = 2, align = 2
    globol powereup, start

    powerup:  clr    0x02fa      /* 开放集成监督定时器 */
              mov.b  0x45d, #0xa5
              mov.b  0x45e, #0x5a
              jmp     start

# endasm
```

其中:

最后的 start 是运行启动模块的入口地址。应该特别指出: 编写上电子程序时, 堆栈和 asume 的一切资源都未就绪, 需要时都得自行安排。

## 15.9 XAC 标准启动模块的编程

### 15.9.1 连接器定义符号名

编写运行启动程序时, 需要用到一些连接器定义的符号名。下面给出定义的规律:

_L 程序子段名	连接时程序子段的起始地址
_H 程序子段名	连接时程序子段的末地址
_B 程序子段名	装载时程序子段的起始地址

[例 15.2]

_Lbss	连接时 bss psect 的起始地址
_Hbss	连接时 bss psect 的末地址

`_Hbss - _Lbss` bss psect 的长度

### 15.9.2 bss 和 rbss 清零程序

下面给出 rbss 子段的清零汇编程序段：

```
clrbbss:
    mov    a, #_Hrbss - _Hrbss
    jz     skip1
    mov    r0, #_Lbss
    mov    r2, a
loop1:
    mov    @r0, #0
    in     cr0
    djnz   r2, loop1
skip1:
```

仿此可写出 bss 子段的清零汇编程序段。如果还有 rbit 子段也可仿此写出。

### 15.9.3 data 和 rdta 复制程序

下面给出 rdata 子段从 ROM 到 RAM 的复制汇编程序段：

```
copyrdata:
    mov    a, #_Hrdata - _Hrdata
    jz     skip3
    mov    dptr, #_Brdta
    mov    r0, #_Lrdta
rdloop:
    clr    a
    movc   a, @a + dptr
    mov    @r0, a
    inc    dptr
    inc    r0
    djnz   r2, rdloop
skip3:
```

仿此可写出 data 子段从 ROM 到 RAM 的复制汇编程序段。

注意：对于 medium 和 large 需要写出上述复制程序段。对于 small 不需要这种复制。

## 15.10 XAC 定制的启动模块

当 ROM 空间不够时，可行的解决办法是：①为了缩短程序可进行人工优化。②删节标准启动模块中用不到的部分。③删节函数库中的版权信息。

### 15.10.1 手工优化代码

手工优化代码是避免过长而低效的代码，侧重于求短代码。措施是：①大、中模式中，关键





变量加 `near` 修饰符,改放到直接寻址的片内 RAM 中去。②选择特别有利的存储模式。若 RAM 为中等规模,尽量试用小模式。③能用 8 位的变量的不要用 16 位,特别是在使用 8 位外部总线时更为突出。

### 15.10.2 定制启动模块的编写

编写定制启动模块的目的是为了删节标准启动模块中本次应用项目用不着的内容,以求获得少量关键的 ROM 空间,使应用项目得以顺利实现。否则,是不值得的。

标准启动模块的源代码在 SOURCE 子目录中可以找到。它的内容包括:清 `rbss`, `bss`, `rbit` 各子段,将 `rdata`, `data` 各子段复制到 RAM。

如果应用程序未用过 `bit` 类型,可以删去清 `rbit` 的汇编程序段。

如果应用程序未用过 `rdata` `psect` 子段中的数据类型,可以删去复制 `rdata` 的汇编程序段。

总之,需要首先研究清楚标准启动模块的源代码,绝对不可冒然删节,不清楚的千万不可乱动。一旦完成了必要的删节,得到的就是定制的源程序。把它放在名为 `RTXAx.AS` 的文件中,用下面的命令行进行汇编:

```
XAS -X RTXA--x.AS
```

其中:

`XAS` 是调汇编器的命令。

`-X` 是调汇编器的选项。

`RTXA--x.AS` 是汇编源程序名,程序名中的 `x` 在大写 `S`, `M`, `L` 中选用,以反映存储模式。

汇编后的定制启动模块 `RTXA--x.OBJ` (注意存储模式!)应放到 `LIB` 目录下(一般是 `C:\HT-XA\LIB`)。LIB 目录应已置入 DOS 环境变量 `LIB` 之内。之后,只要重新连接一遍,模式一致的定制启动模块自然会被选中,不需再做任何其他的工作。

使用了定制启动模块的应用程序,最好不要再改动。若必须改动,要非常小心。若当前使用的是最小版本的定制启动模块,只要后来添加了被放在 `bss`, `data`, `rbit` 的变量就会发生问题。由于该清零的未清零,该复制的未复制,程序的运行不会稳定,甚至每次重新运行得到几乎是随机的结果。

### 15.10.3 关于版权信息

在标准启动模块中还有一段关于 `HI-TECH C` 库函数的版权信息,它被连接在 ROM 的开始部分。删除这段版权信息,可以获得 80 个字节的 ROM 空间。

## 第十六章 XAC 的混合编程和函数库

### 16.1 C 语言与汇编语言混合编程

XAC 提供两种 C 与汇编混合编程的方法:函数的交叉调用和在线汇编指令段。

#### 16.1.1 C 与汇编函数的交叉调用

不同的语言只要按照统一规范定义函数和调用函数,就可以进行交叉调用。规范正是不同语言之间交叉调用的接口。C 语言有明确的函数定义和函数调用规范,严格的参数和返回规定,所以,只要编写汇编子程序和调用 C 语言的函数时,考虑到还有参数和返回规定,就能够实现语言函数间的交叉调用。C 语言有外部调用前的函数引用说明,同样,汇编语言也有外部函数调用前相应的伪指令说明。

##### 1. C 中调汇编子程序

C 语言中:

extern 类型说明符 汇编子程序名(形参表);  
汇编子程序名(实参表);

汇编语言中:

按汇编语言格式写子程序,但要考虑:

- (1) 按 C 的参数传送规定以取得实参数。
- (2) 在 RET 前按 C 的返回量规定将它放入适当寄存器中。
- (3) 求出函数的签字值,并放入伪指令 SIGNAT 之后。

##### [例 16.1] C 中调汇编左移位子程序

C 语言中:

```
extern char rotate_left(char c)
rotate_left(c);
```

汇编语言中:

```
PSECT TEXT, CLASS=CODE
GLOBAL _ROTATE_LEFT
SIGNAT _ROTATE_LEFT, 4201
PSECT TEXT
_ROTATE_LEFT:
    MOV R0, R3
    RL R0
    RET
```

说明:

(1) 汇编中的标号是子程序的入口地址或 C 中的函数名。在 C 的内部给函数名一律加前导下划线。所以,汇编的标号要加前导下划线。

(2) 汇编中的伪指令 SIGNAT 之后要放子程序名和函数的签字值。函数的签字值前面介绍过求法。

(3) 请注意参数的传入和返回量的传出方法。



## 2. 汇编中调 C 的函数

C 语言中:

写被调函数的定义性说明

汇编语言中:

- (1) 按 C 的规定用寄存器或堆栈向函数传入参数。
- (2) 使用合适的 CALL 指令。
- (3) 等被调函数返回后, 由合适的寄存器取出返回量进行处理。

超星浏览器提醒  
使用本复制品  
请尊重相关知识产权!

### 16.1.2 在线嵌入汇编指令段

C 中在线嵌入汇编指令有两种方法: 一次嵌入一条汇编指令和一次嵌入一段汇编指令。

(1) 嵌入一条汇编指令的格式为:

```
asm(汇编指令);
```

(2) 嵌入一段汇编指令的格式为:

```
#asm
```

```
...
```

```
#endasm
```

[例 16.2]

```
#include <stdio.h>
unsigned char var;

void main( )
{
    var = 1;
    printf("var = 0x%2.2X\n", var);
    #asm
        mov r0, _var
        rl r0
        mov _var, r0
    #endasm
    printf("var = 0x%2.2X\n", var);
    asm(mov r0, _var);
    asm(rl r0);
    asm(mov _var, r0);
    printf("var = 0x%2.2X\n", var);
}
```

## 16.2 XAC 运行时间库函数

XAC 库函数是随 XAC 编译器一同提供的运行时间库。

### 16.2.1 标准输入输出库函数及用户的定制

就微控制器的运行时间库而言, 与 ANSI C 的不同要算是标准输入输出库函数了。ANSI C 的标准输入输出库中用的设备指的是控制台, 即字符输入设备为键盘, 字符输出设备为显示器。XAC 提供的标准输入输出库用的是串行口, 即字符的输入输出均通过串行口。表 16.1

列出 XAC 的标准输入输出库函数。

表 16.1 XAC 的标准输入输出库函数

函数名	功 能
puts(char * s)	向标准输出写字符串, 等下行
gets(char * s)	从标准输入读一行字符串, 等下行
printf(char * fmt, ...)	读参数格式化后写到标准输出
putchar(int c)	标准输出写字符
scanf(char * fmt, ...)	按格式从标准输入读字段格式化后写到地址参数
sprintf(char * buf, char * fmt, ...)	读参数格式化后写到字符串
sscanf(char * buf, char * fmt, ...)	按格式从字符串读字段格式化后写到地址参数
vprintf(char * fmt, va_list arglist)	从变参数表读内容格式化后写到标准输出
vscanf(char * fmt, va_list arglist)	按格式从标准输入读字段格式化后写到变参数表

如果所开发的目标机不是使用串行口输入输出字符, 有两个库函数源文件可能需要修改——getch.c 和 serial.c。前者包括 getch(), getche(), putch(), kdhit(); 后者是串行口的底层驱动程序。它们可以在 SOURCE 子目录中找到。修改时, 最好先复制到工作目录中, 再做修改。修改后, 先编译再连接。连接时, 将它(们)作为目标文件放入命令行即可, 即它(们)会代替原库中同名函数被连入。如果使用 HI-TECH 的 51XA 开发平台 HPDXA 就更简单了, 把修改后的源文件放入工程项目(project), 然后直接执行实用程序 make 即可。

### 16.2.2 XAC 库函数汇总

表 16.4 分类列出 XAC 提供的库函数名。库函数的详细解释和用法请见附录 E。附录 E 按字母排列, 以便查找。

表 16.4 XAC 库函数汇总表

函数名	所在头文件
1. 数学函数	
fabs	math.h
exp	math.h
log	math.h
log10	math.h
sqrt	math.h
rand	stdlib.h
srand	stdlib.h
cos	math.h
sin	math.h
tan	math.h
acos	math.h
asin	math.h
atan	math.h
cosh	math.h
sinh	math.h
tanh	math.h

续表 16.4

函数名	所在头文件
ceil	math.h
floor	math.h
pow	math.h
DIV	stdlib.h
LDIV	stdlib.h
LDEXP	stdlib.h
FREXP	stdlib.h
2. 标准输入输出函数	
gets	stdio.h
printf	stdio.h
sprintf	stdio.h
puts	stdio.h
scanf	stdio.h
getc	conio.h
getch	conio.h
getch	econio.h
ungetch	conio.h
cgets	conio.h
putch	conio.h
cputs	conio.h
vprintf	stdio.h
vscanf	stdio.h
kdhit	conio.h
3. 动态存储函数	
calloc	stdlib.h
free	stdlib.h
malloc	stdlib.h
realloc	stdlib.h
4. 字符归类函数	
isalpha	ctype.h
isalnum	ctype.h
isctrl	ctype.h
isdigit	ctype.h
isgraph	ctype.h
isprint	ctype.h
ispunct	ctype.h
islower	ctype.h
isupper	ctype.h
isspace	ctype.h
toascii	ctype.h
tolower	ctype.h
toupper	ctype.h

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

续表 16.4

函数名	所在头文件
isascii	ctype.h
5. 字符串函数	
memchr	string.h
memcmp	string.h
memcpy	string.h
memmove	string.h
memset	string.h
strcat	string.h
strncat	string.h
strncmp	string.h
strlen	string.h
strchr	string.h
strrchr	string.h
6. 字符串转换函数	
atof	stdlib.h
atol	stdlib.h
atoi	stdlib.h
7. 参数函数	
va_start	stdarg.h
va_arg	stdarg.h
va_end	stdarg.h
8. 全程跳转函数	
setjmp	setjmp.h
longjmp	setjmp.h
9. 时间函数	
asctime	time.h
ctime	time.h
gmtime	time.h
localtime	time.h
time	time.h
10. 中断类函数	
setvector	intrpt.h
di	intrpt.h
ei	intrpt.h
11. 系统函数	
persist_check	sys.h
persist_validate	sys.h
exit	stdlib.h
bsearch	stdlib.h
assert	stdlib.h

超星阅读器提醒您：  
使用本复制品  
请尊重相关知识产权！

### 16.2.3 XAC 库管理器实用程序

库管理器实用程序 LIBR 的功能是专供用户把多个目标文件合并为一个可放在自定义库中的库文件。把多个目标模块合并的目的有：

- 减少连接时的文件数目。
- 加快存取速度。
- 节省磁盘空间。



库是关于目标文件的一种特殊管理形式。根据库是连接器在连接时专用的特点，连接器对于库文件不是像连接普通目标文件那样全盘连入，而是按照连接器可识别的模块和符号仅提取其必要的部分连入。库中的搜索采用线性方法，因此，库中模块的安放次序会影响连接结果。

#### 1. 库格式

库中的各个模块仅是简单的接续安放，但在库的开始部分有一个由模块和符号组成的目录。由于目录远小于模块的总和，因此使搜索速度得以提高。读取的又只是模块的一部分，对磁盘的 I/O 操作削减了许多。

#### 2. LIBR 库的使用

LIBR 库的命令行格式为：

```
LIBR k file.lib [file1.obj [,obj2 ...]]
```

其中：

- (1) k——库使用方式关键字，见表 16.5。

表 16.5 LIBR 库使用方式关键字汇总表

库使用关键字	意 义
r	替换模块
d	删除模块
x	提取模块
m	模块列表
s	模块列表，包括全局符号

- (2) file.lib——库文件名。

- (3) [file1.obj [,obj2 ...]]——0 到多个模块名。

- (4) 选项 r, x——使用替换和提取关键字 r, x 时，模块名是操作的对象，未指定模块名时全部模块名都是操作的对象。

- (5) 选项 r——使用替换关键字 r 时，所指定的文件名或模块名。若库中没有时，则创建之。创建的内容总是接续在相应的已存在内容的最后。如果是替换，换后位置不变。

- (6) 选项 d——使用删除关键字 d 时，模块名是操作的对象，未指定模块名时报错。

- (7) 选项 m, s——使用列表关键字 m, s 时，模块名是操作的对象，未指定模块名时全部模块名都是操作的对象。m 与 s 的区别是：后者同时列出全局符号。列表的内容输出到标准输出设备上，每个模块名占一行。要求同时列出全局符号的，在各模块名后列出全局符号名，并在名字前加 D 或 U 表示定义或引用。

- (8) 模块名次序——模块名在库中次序对连接器的连接有重要意义。如果一个模块引用

同一库文件中的另一模块的符号,应将定义模块放在引用模块的后面。

(9) 命令行超长——LIBR 的命令行最长不得超过 127 个字符。由于 LIBR 命令行要输入很多的目标文件,经常不够使用。解决办法有二:①在逻辑行的末尾使用续行符‘\’;②使用扩展名为 .cmd 的命令文件。命令文件中存放命令行上除 LIBR 之外的其他全部内容。命令文件经重定向向命令行输入其内容。使用命令文件的命令格式为:LIBR < file.cmd

[例 16.3] 列出模块名和全局符号。

```
LIBR s file.lib a.obj b.obj c.obj
```

[例 16.4] 删库文件中的指定模块。

```
LIBR d file.lib a.obj b.obj c.obj
```

[例 16.5] 使用续行符。

```
LIBR r file.lib a.obj b.obj c.obj \  
d.obj e.obj f.obj
```

[例 16.6] 使用命令文件。

```
LIBR < file.cmd
```

命令文件 file.cmd 的内容为:

```
r file.lib a.obj b.obj c.obj  
d. obj e.obj f.obj
```

使用本复制品  
请尊重相关知识产权!

# 第十七章 XAC 编译器



## 17.1 编译命令行控制选项

编译器的首要任务是生成 ROM 代码。XAC 编译命令行的编译控制选项见表 17.1。这里介绍的编译控制选项包括高级程序员使用的专业控制选项。一般程序员使用后面介绍的 HPDXA 集成开发环境即可。

表 17.1 XAC 编译控制选项表

控制选项	意 义
- Aspec	为连接/定位指定存储器地址
- AAHEX	生成美国自动化 HEX 符号表
- ASMLIST	为每次汇编生成列表文件 .LST
- AV	生成 AVOCET 格式的符号表
- BIN	生成二进制输出文件
- Bl	选用大存储模式
- Bm	选用中存储模式
- Bs	选用小存储模式
- C	只编译生成目标文件
- CLIST	生成 C 原程序二进制列表文件
- CRfile	生成交叉访问表
- Dmacro	在命令行上定义预处理器宏
- DOUBLSE	指定 DOUBLE 为 64 位浮点数
- E	编译错使用 EDITOR 格式
- Efile	编译错重定向到指定文件
- E+ file	编译错附加到指定文件之后
- Gfile	生成源级符号表
- Hfile	生成汇编级符号表
- HELP	显示控制选项表
- Ipath	指定 #include 文件所在目录的路径名
- Llibrary	指定被连接器扫描的库文件名
- P - option	为连接器直接指定控制选项
- Mfile	生成映像文件
- MOTOROLA	生成 MOTOROLA HEX 格式的输出文件
- Nnum	指定标识符的缺省长度
- O	指定后趟优化
- Ofile	指定输出文件名和类型
- OMF51	生成 OMF51 输出文件
- PROTO	生成新的和过时的函数原形说明格式
- PSECTMAP	显示连接后全部的程序段映像



续表 17.1

控制选项	意 义
- S	只编译到汇编源程序文件
- SA	编译到 Avocet AVMAC 汇编源程序文件
- STRICT	指定符合 ANSI C 规定的关键字格式
- TEK	生成 Tektronic HEX 格式的输出文件
- Umacro	释放预处理器定义宏
- UBROF	生成 UBROF HEX 格式的输出文件
- UNSIGNED	指定 UNSIGNED CHAR 为 CHAR 的缺省类型
- V	显示编译命令行的内容
- Wlevel	设置编译警告级别
- X	从符号表中消去局部符号
- Zg	启动全局优化

### 17.1.1 - A(指定 ROM 和 RAM 定位地址)

格式:

- A rom, ram, ramsize, nvram, bankram

其中:

(1) rom——指定程序的起始地址。对于 XA 它是 0。由 ROM 的 0 地址开始,首先是复位向量,其次是中断向量表。如果为了使用 LUCIFER 调试器而编译,取下装代码到 RAM 的首地址。

(2) ram——指定数据使用片上 RAM 的起始地址。它不应小于 20H。小于 20H 是通用寄存器的影子。从 20H 开始,先是 20H 字节的按位寻址空间,以后是字节寻址空间。20H 是缺省值。

(3) ramsize——使用片上 RAM 的大小。缺省值是 1E0H。它与通用寄存器共占 200H 片上 RAM 空间。XA 的片外 RAM 从 200H 开始接续编址。

(4) nvram——persistent 变量使用的非易失性 RAM 地址。如果全部 RAM 都是非易失性的或没有 persistent 变量, nvram 必须给 0 值。缺省值是 0。

(5) bankram——按体(64k)分配给 far 变量的片外 RAM 地址。它必须是一个体的基地址。缺省值是 10 000H。如果没有 far 变量,本地址不必要。

上述各量必须使用十六进制数,并且不用后缀 H。

[例 17.1] - A0,20

ROM 由 0 地址开始, RAM 由 20H 开始。RAM 长度为缺省值 1E0H, 没有 persistent 变量和 far 变量。

[例 17.2] - A0,20,1FD0,1FF0,20 000

ROM 由 0 地址开始, RAM 由 20H 开始, RAM 长度为 1FD0H, persistent 变量放在由 1FF0H 开始的非易失性 RAM 中, far 变量放在由 20 000H 开始的片外 RAM 体中。

### 17.1.2 - AAHEX(指定按美国自动化符号格式生成 HEX 文件)

按美国自动化符号格式生成 HEX 文件,它是 Motorola S-Record 格式文件的扩展格式。

它是在 Motorola S-Record 格式文件的开头加上符号记录形成的。如果使用美国自动化在线仿真器调试时,必须加本选项进行编译。本选项只对扩展名为 .HEX 的文件有效,对扩展名为 .BIN 的文件无效。

### 17.1.3 -AV(指定符号文件用 Avocet 风格)

本选项指定符号文件用 Avocet 风格,但不产生符号文件。必须与 -H 选项连用才产生名叫 test.sym 的 Avocet 风格符号表。test.sym 的 Avocet 风格符号表是用于 Avocet Systems 的仿真器和 AVSIM0。

[例 17.3] XAC -AV -Htest.sym -A0,30,100 test.c

### 17.1.4 -BIN(指定生成二进制输出文件)

指定生成的二进制输出文件具有扩展名 .bin。

### 17.1.5 -BI(指定选用大存储模式)

大存储模式使用 XA 的远调用和远返回,访问空间因芯片而异,最大 16MB。ROM 空间按 64KB 为一体,顺序按体选用。RAM 空间最大 64KB,外加 far RAM 空间。

尽管大存储模式的函数地址是 24 位,但是,函数指针是 32 位的。大存储模式所用的启动模块和运行时间库见表 17.2。

表 17.2 大存储模式所用的启动模块和运行时间库表

启动模块和运行时间库	用 途
RTXA- -L.OBJ	大存储模式所用的启动模块
XA- -LC.LIB	大存储模式标准库,32 位 double
XA- -LL.LIB	大存储模式 printf 库,支持 long
XA- -LF.LIB	大存储模式 printf 库,支持 long 和 float
XA- -DLC.LIB	大存储模式标准库,64 位 double
XA- -DLL.LIB	大存储模式 printf 库,支持 long, double
XA- -DLF.LIB	大存储模式 printf 库,支持 float, double

### 17.1.6 -Bm(指定选用中存储模式)

中存储模式 ROM 空间最大 64 KB, RAM 空间最大 64 KB,外加 far RAM 空间。

中存储模式的函数指针是 16 位的。变量用间址或变址寻址方式,修饰为 near 的变量位于片内(400H),用直接寻址方式访问。

中存储模式所用的启动模块和运行时间库见表 17.3。

表 17.3 中存储模式所用的启动模块和运行时间库表

启动模块和运行时间库	用 途
RTXA- -M.OBJ	中存储模式所用的启动模块
XA- -MC.LIB	中存储模式标准库,32 位 double
XA- -ML.LIB	中存储模式 printf 库,支持 long

续表 17.3

启动模块和运行时间库	用 途
XA - -MF.LIB	中存储模式 printf 库, 支持 long 和 float
XA - -DMC.LIB	中存储模式标准库, 64 位 double
XA - -DML.LIB	中存储模式 printf 库, 支持 long, double
XA - -DMF.LIB	中存储模式 printf 库, 支持 float, double

### 17.1.7 -Bs(指定选用小存储模式)

小存储模式是缺省模式。小存储模式的 ROM 空间最大 64 KB。RAM 空间最大 1 KB。

小存储模式的函数指针是 16 位的。变量一律用直接寻址方式访问。带初值的 static 变量和字符串常量放在 ROM 中, 用 MOVC 指令来存取。带初值的 static 变量运行中不可改变。

小存储模式所用的启动模块和运行时间库见表 17.4。

表 17.4 小存储模式所用的启动模块和运行时间库表

启动模块和运行时间库	用 途
RTXA - -S.OBJ	小存储模式所用的启动模块
XA - -SC.LIB	小存储模式标准库, 32 位 double
XA - -SL.LIB	小存储模式 printf 库, 支持 long
XA - -SF.LIB	小存储模式 printf 库, 支持 long 和 float
XA - -DSC.LIB	小存储模式标准库, 64 位 double
XA - -DSL.LIB	小存储模式 printf 库, 支持 long, double
XA - -DSF.LIB	小存储模式 printf 库, 支持 float, double

### 17.1.8 -C(只翻译到目标文件)

本选项使编译器编译到目标文件就终止。本选项允许在命令行上接受 .c, .as 或 .obj 文件的任意组合。遇到 .c 文件则编译, 遇到 .as 文件则汇编, 遇到 .obj 文件则留待连接时再用。本选项在 make 实用程序中有很好的应用。

[例 17.4] XAC -O -Zg -C main.c module1.c asmcode.as

### 17.1.9 -CR(生成交叉访问表)

格式:

-CR[文件名]

其中:

文件名是指定存放交叉访问表的文件。文件名可有可无。如果没有文件名, 则生数据留在暂存文件中, 等待实用程序 CREF 处理为正式交叉访问表文件。如果有文件名, 则 XAC 自动启动实用程序 CREF 得到交叉访问表并放于文件中。

[例 17.5] XAC -CRmain.crf main.c module.c nvram.c

### 17.1.10 -CLIST(生成 C 的列表文件)

为每一个 C 源文件生成一个带行号等的列表文件。列表文件名是命令行上第一文件主

名加 .LST。

### 17.1.11 -D(定义宏)

格式:

-D 宏名 [= 宏体]

其中:

宏名和宏体与预处理器中的定义一样。宏体可有可无。无宏体时宏体内容固定为 1。

[例 17.6] XAC -Ddebug -Dbuffers=10 test.c

在编译 test.c 的同时定义了两个预处理器宏, 如同:

```
#define debug 1
#define buffers 10
```

### 17.1.12 -DOUBLE(起用 IEEE 64 位 DOUBLE 变量)

XAC 的缺省 double 值是 32 位。本选项起用 IEEE 64 位 DOUBLE 变量。

### 17.1.13 -E(编译器使用 editor 格式的错误信息)

XAC 使用由脱字符直接指出语句行上错误的标准错误信息格式。而 editor 指出错误的风格是先给出错误所在的文件名、行号和列数, 然后指出错误性质。

无论是标准格式还是 editor 风格, 都可以进行输出的重定向。如:

XAC -E x.c > errlist

将 x.c 的编译错误信息重定向到 errlist 中。也可以将错误信息附加到重定向的文件之后, 如果文件不存在则创建之。如:

XAC -E x.c >> errlist

### 17.1.14 -E(编译器错误信息重定向到指定文件)

格式:

-E[+]文件名

其中:

(1) 文件名——是错误信息重定向到的文件。

(2) + 号——可有可无。有 + 号是将错误信息附加到重定向文件。

[例 17.7]

XAC -E project.err -O -Zg -C main.c

XAC -E +project.err -O -Zg -C part1.c

XAC -E +project.err -C asmcode.as

project.err 文件包含三个源文件的错误信息。譬如:

main.c 11 23: | expected

main.c 18 33: ; expected

part1.c50: type redeclared

part1.c50: argument list conflict with prototype

asmcode.as 22 0: Syntax error



### 17.1.15 -H(生成汇编级符号文件)

格式:

-H 文件名

其中:

文件名——生成的汇编级符号放本文件中。HI-TECH 的软件调试器只能做汇编级调试。本选项与 -AV 选项联合可生成 Avocet 风格的汇编级符号表。

### 17.1.16 -I(指定附加的搜索头文件的路径)

格式:

-I 路径名

其中:

- (1) 路径名——指定搜索头文件的路径。
- (2) 本选项在命令行上可多次使用。
- (3) 本选项缺省时,搜索标准头文件的路径。

[例 17.8]

```
XAC -C -Ic:\include -Id:\myapp\include test.c
```

### 17.1.17 -L(指定附加的扫描库)

格式:

-L 附加的扫描库关键字

其中:

- (1) 附加的扫描库关键字——它是与标准库名相区别的关键字(见表 4.2~表 4.4)。
- (2) 附加库的函数与标准库函数重名时,附加库的函数将替代标准库中的重名函数。

[例 17.9] 在小模式中使用浮点版本的 printf() 替代标准库中的版本。

```
XAC -LF test.c
```

使用 -LF 选中 XA-SF.LIB。

### 17.1.18 -L- (指定传递给 LINKER 的控制选项)

格式:

-L- LINKER 的控制选项

其中:

-LINKER 的控制选项——打算传递给 LINKER 的控制选项。

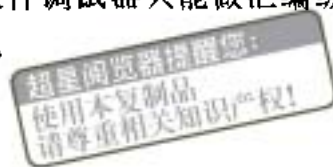
[例 17.10] 将 LINKER 的控制选项 -FOO 提前在 XAC 命令行上给出。

```
XAC -L-FOO test.c
```

[例 17.11] 将 LINKER 定位特殊程序子段(psect)的控制选项提前在 XAC 命令行上给出。

```
XAC -Bl -L-Pxram=1800h -A0,30,8000,4000 prog.c xram.c
```

-P 是 LINKER 的控制选项, xram 是定位于已于 1800H 的特殊程序子段(psect)。



### 17.1.19 -M(生成映像文件)

格式:

-M[文件名]

其中:

文件名——指定存放映像表的文件名。它可有可无。没有时,缺省文件名是 1.map。

### 17.1.20 -MOTOROLA(生成 Motorola S-Record 格式的 HEX 文件)

指示 XAC 生成 Motorola S-Record 格式的 HEX 文件。只对扩展名为 .HEX 的文件有效,对 .BIN 无效。

### 17.1.21 -N(指定标识符有效字符长度)

格式:

-N 长度

其中:

长度——指定标识符有效字符长度。缺省值是 ANSI/ISO 标准规定的 31。长度可选范围是 31~255。

### 17.1.22 -O(启动优化)

本选项启动窥孔优化,消除冗余跳转和寄存器装入指令,以减少代码长度。

### 17.1.23 -O(指定输出文件)

格式:

-O[文件名]

其中:

文件名——指定 XAC 使用的输出文件名。指定输出文件名时应注意扩展名,如: .HEX, .BIN, .UBR。它们分别产生不同类型的输出文件。本选项可有可无。缺省时,取第一个文件的主名加其他扩展名。

[例 17.12] XAC -Otest.bin -A0,30,2 000, prog1.c part2.c

指定 XAC 输出二进制文件 test.bin。

### 17.1.24 -OMF51(指定生成 OMF51 格式的输出文件)

指定生成 OMF51 格式的输出文件。它是一个带调试信息的可执行代码文件。它只支持 64 KB 代码空间和 64 KB 数据空间。本格式可在许多仿真器上运行。

### 17.1.25 -PROTO(指定生成包括 ANSI 和 K&R 风格的函数原型文件)

本选项指定生成包括 ANSI 和 K&R 风格的函数原型文件(.pro)。

[例 17.13] 使得用了 K&R 风格的源文件也能生成包括 ANSI 和 K&R 风格的函数原型文件。下面是 test.c 源文件。



```
#include <stdio.h>
int add(arg1, arg2)
int * arg1, arg2;
{
    return * arg1 + * arg2;
}
void printlist(list, count)
int * list, count;
{
    while(count --)
        printf("%d", * list ++);
    putchar('\n');
}
```



编译如下：

```
XAC -proto test.c
```

产生如下内容的 test.pro:

```
#if PROTOTYPES
extern int add(int *, int *);
extern void printlist(int *, int);
#else
extern int add();
extern void printlist();
#endif
```

有了 test.pro 的内容就可以组织头文件了。

### 17.1.26 -PSECTMAP(程序段映像表)

本选项要求给出代码全部连接之后的存储器 and 程序子段的映像表。它比存储器映像表更详细。

### 17.1.27 -S(编译生成汇编源文件)

本选项要求编译器仅生成汇编源文件,之后终止。生成汇编源文件时,可以加优化选项, -O、-Zg。本选项常用于了解函数调用规则和签字值。

[例 17.14] XAC -O -Zg -S test.c

### 17.1.28 -STRICT(严格遵守 ANSI 标准)

本选项要求所有的特殊关键字都符合 ANSI 标准。XAC 中的 near, far 和 interrupt 不符合 ANSI 的 -near, -far, -interrupt, 为了严格符合 ANSI 标准应加 -STRICT。



### 17.1.29 -TEK(编译生成 Tektronics HEX 文件)

编译生成 Tektronics HEX 文件。只对 .HEX 文件有效,对 .BIN 文件无效。

### 17.1.30 -U(解除宏定义)

格式:

-U 宏名

其中:

- (1) 宏名——曾经定义过的宏,包括预定义宏。
- (2) 它是 -D 的逆。

### 17.1.31 -UBROF(指定生成 UBROF 格式的输文件)

指定生成 UBROF 格式的输文件。有不少仿真器使用这种格式。本选项生成的输文件具有 .UBR 扩展名。选项 -O 加具有 .UBR 扩展名的输文件与本选项作用一样。

### 17.1.32 -UNSIGNED(指定 unsigned char 为 char 的缺省类型)

XAC 原来指定 char 的缺省类型是 signed char。加选项 -UNSIGNED 后 char 的缺省类型改为 unsigned char。这时,对于符号量要显式地使用 signed char。

### 17.1.33 -V(详示编译命令)

指定 XAC 每次编译都要显示出相应的命令行。本选项对于了解准确的连接命令以便直接启动连接器非常有用。

### 17.1.34 -W(设置告警级别)

格式:

-W 级别

其中:

级别——告警级别。范围是 -3~9。缺省级别是 0,给出正常告警信息。-3 挑剔最多,9 将所有告警信息全部都抑制掉。

[例 17.15] XAC -W-2 -C foo.c

### 17.1.35 -X(去除局部符号)

从各个编译、汇编和连接文件中去除局部符号,仅在各个目标文件和符号文件中保留全局符号。

### 17.1.36 -Zg(启动全局优化)

在代码生成阶段启动全局优化。它能大量地缩减代码长度和片上 RAM 的用量。



## 17.2 编译器输出文件格式

表 17.5 列出 XAC 可以生成的多种输出文件格式。通过第三列的编译命令行选项可以直接生成相应格式的输出文件。编译命令行上没有第三列所列的选项,则作为缺省,生成 Intel 格式的 HEX 输出文件,它广泛应用于通用的 PROM 写入器和许多在线仿真器。还可以用编译命令行选项 -O 生成与上述同样的输出文件。不过,这时需要附加输出文件名,文件名的扩展部必须按表 17.5 的第四列选用。

表 17.5 输出文件格式

输出文件名	说 明	XAC 选项	文件类型
Motorola HEX	s1/s9 型 hex 文件	- Motorola	.HEX
Intel HEX	Intel 风格 hex 记录格式	缺省	.HEX
Binary	一般 binary 影像文件	- BIN	.BIN
UBROF	通用二进制可再定位影像文件	- UBROF	.UBR
Tecktronix HEX	Tecktronix 风格 hex 记录	- TEX	.HEX
American Automation HEX	美国自动化带符号 hex 记录	- AAHEX	.HEX
OMF-51	Intel 绝对目标模块格式	- OMF	.OMF

## 17.3 编译器生成的符号文件

XAC 有三个选项涉及到生成的符号文件,即: -G, -H 和 -AV。

-H 只生成汇编级调试用符号文件, -G 则生成包括 C 语言源级调试用信息和汇编调试信息的符号文件。-G 和 -H 都要求文件名作为其参数。如果未给文件名,则用 1.sym 作为缺省文件名。使用 -G 和 -H 选项生成的符号文件,可用于在线仿真器和模拟器进行源级和汇编级的调试。Nohau 和 Ashling 生产 51XA 的在线仿真器,可以与 XAC 配套使用。

[例 17.16] 试生成 C 语言源级调试用信息的符号文件。

XAC -Gtest.sym test.c

另外, -H 和 -AV 连用则生成 Avocet 风格的汇编级调试用符号文件,用于 Avocet System 生产的仿真器和模拟器。

## 17.4 CREF 生成交叉访问表的实用程序

CREF 将编译器和汇编器生成的生交叉访问信息进行处理和排序得到通常所说的交叉访问表。CREF 命令行格式如下:

CREF 选项表 文件表

其中:

- (1) 选项表——见表 17.6 CREF 选项表。
- (2) 文件表——是由编译器和汇编器生成的包含生交叉访问信息的各个文件名。
- (3) CREF 可以接受通配符文件名。

(4) CREF 可以接受 I/O 重定向。

表 17.6 CREF 选项表

选 项	意 义
-F 路径或文件名	排除指定文件(路径)中的各种符号
-H 表头名	指定列表文件的表头名
-L 每页行数	指定列表文件每页行数
-O 输出文件名	指定列表文件名
-P 页宽	指定列表文件页宽
-S 包含拒选符号的文件名	指定列表文件的拒选符号构成的文件名
-X 拒选符号的前导字符序列	指定列表文件拒选的具有指定前导字符序列的各个符号

#### 17.4.1 -F 路径或文件名

经常希望在交叉访问表中除去某些头文件中的符号,如<stdio.h>,这时就需要用选项 -Fstdn.h。如果想除去整个目录下全部文件的符号,就要用路径名。如:-F 则将根目录下的各个文件的符号都排除掉。

#### 17.4.2 -H 表头名

为交叉访问表起表头名。缺省表头名取第一个生交叉访问信息文件名。

#### 17.4.3 -L 每页行数

指定列表文件每页行数。缺省值为 66 行。

#### 17.4.4 -O 输出文件名

指定列表文件名。缺省时输出到标准输出设备上,允许重定向。

#### 17.4.5 -P 页宽

指定列表文件页宽。缺省值为 88 字符。宽行打印取 132。

#### 17.4.6 -S 包含拒选符号的文件名

指定列表文件中由拒选的符号构成的文件名。本选项可以多次选用,以列出多个包含不同拒选符号的文件名。

#### 17.4.7 -X 拒选符号的前导字符序列

指定列表文件拒选的具有指定前导字符的各个符号,例如,-Xxyz,则所有以 xyz 开始的符号都在拒选之列;又-XX0,则所有 X 后跟数字的符号都在拒选之列。

## 第十八章 XAC 预处理器

使用本复制品  
请尊重相关知识产权!

下面只介绍 XAC 预处理器的特有部分。

### 18.1 XAC 预定义宏

XAC 的驱动程序为预处理器(CPP)的条件编译预定义了一些宏,见表 18.1。它们均定义为 1。

表 18.1 XAC 预定义宏

预定义宏	用 途
HI-TECH-C	已被置位,表示所用编译器为 HI-TECH-C
-XA-	已被置位,表示为芯片 51XA 而编译
SMALL-MODEL	如果置位,表示是小模式
MEDIUM-MODEL	如果置位,表示是中模式
LARGE-MODEL	如果置位,表示是大模式

### 18.2 #pragma 编译控制伪指令

有一些写在源文件中的编译伪指令可以改变编译器的行为。按 ANSI 标准这时应使用预处理器伪指令 #pragma, 格式如下:

#pragma 关键字 [选项]

其中:

- (1) 关键字——可用的关键字见表 18.2。
- (2) 选项——有的关键字需要选项作为它的参数,有的不要求。详见表 18.2 第一列。

表 18.2 pragma 伪指令

伪指令关键字及选项	意 义	示 例
jis	起用双字节字符管理程序	#pragma jis
nojis	释放双字节字符管理程序	#pragma nojis
pack 1	指定在结构中不使用字对齐	#pragma pack 1
printf-check(函数名)	有些变参数函数使用格式字符串,如果要求编译时对变参数进行一致性检查应加本伪指令	#pragma printf-check(printf)
psect 编译段名 = 重定位段名	将指定段进行指定地重定位	#pragma psect text = mytext
strings 修饰符	为匿名字符串加修饰符	#pragma strings code

续表 18.2

伪指令关键字及选项	意 义	示 例
switch auto direct simple	switch 语句有 direct 和 simple 两种生成代码的方法,一个速度快些一个代码短些。auto 为恢复以前的设置。本伪指令只对本函数的紧跟的第一个 switch 语句有效。	#pragma switch direct

# 第十九章 XAC 宏汇编器

使用本复制品  
请尊重相关知识产权!

## 19.1 序 言

XA 宏汇编器(ASXA)用于对 51XA 微控制器的汇编源文件作汇编,以取得目标码。XA 宏汇编器完整的套件包括:宏汇编器、连接器、库管理器、交叉访问处理器和目标代码转换器。除宏汇编器外,其他已在不同的地方讲过。

关于 51XA 微控制器的指令集和它的特殊功能寄存器符号请见 51XA 微控制器的手册。

## 19.2 XA 汇编源文件语句

汇编源文件由汇编指令语句和汇编伪指令语句构成。下面介绍这两种语句。先介绍构成汇编语句的单词,再介绍句法。

### 19.2.1 字符集

除操作码和保留字外,标识符使用 7 位的标准 ASCII 字符集。TAB 字符等价于多空格符。

### 19.2.2 数

汇编语言的数学运算一律用 32 位符号数。超范围的数便产生误差。数可用数制和标注见表 19.1。

表 19.1 可用数制和标注

数 制	标 注
二进制	加后缀 B, b 不能用,因 ASXA 中用于暂时标号
八进制	加后缀 O, o, Q, q
十进制	加后缀 D, d 或无
十六进制	加后缀 H, h 并要数字打头,或加前缀 0x

只有 DF 伪指令可以使用实型数。要用十进制数写实型数的尾数和指数部分。实型数可以只用带小数点的十进制数而无指数部分。实型数在表示为指数时,指数与前面的尾数之间不能有空格。内部存放实型数使用 IEEE 的 32 位格式。

### 19.2.3 分隔符

数与标识符用空白符、非字母数字或一行的结束作为分隔符。

### 19.2.4 特殊字符

在宏体中用@作为单词的接续符。参数宏的参数用< >括起。

### 19.2.5 标识符

标识符是用户定义的符号,用以表示地址或数。用于标识符的字符包括:字母、数字、\$、?和\_。标识符不能用数字打头。ASXA 中的标识符是大小写敏感的。

### 19.2.6 汇编生成的标识符

使用了 LOCAL 伪指令的宏块,汇编器为宏而展开时对每一个确定的标识符生成?? nnnn 形式的唯一标识符。用户要避免使用这一形式的标识符。

### 19.2.7 位置计数器

当前 PSECT 子段的位置可以用 \$ 号进行存取。

### 19.2.8 寄存器符号

寄存器可以用它的保留字来表示,保留字是大小写不敏感的。寄存器的保留字不可以用做宏的参数。但是,寄存器的保留字可以用 EQU 伪指令指定为标识符,这时将是大小写敏感的,并可用做宏的参数。

### 19.2.9 字符串

字符串是放在引号内的不包括回车和换行符的字符序列。在 ASXA 中引号可以是单引号对或双引号对。字符串作为伪指令 DB 的操作数可以任意长度,作为指令的操作数只能是 1~2 个字符。

### 19.2.10 暂时标号

为了区别于宏块中的局部标号,给代码块规定了暂时标号,以解除标号重名的顾虑。暂时标号使用纯数字串,而访问标号时,则加 b 或 f 的后缀,b 提示向回找,f 提示向前找。

[例 19.1]

entry:

```
    mov    a, plot
1:    mov    a, @r0
    jz     1f
    inc    r0
    cjne   a, r2, 1b
    sjmp   2f
1:    clr    a
    ret
2:    dec    r0
    mov    a, r0
```



ret

虽然有两个标号 1, 但不会发生混乱。一个由 jz 1f 向前找, 一个由 cjne a, r2, 1b 向回找。

### 19.2.11 表达式

表达式由数、符号、字符串经运算符连接而成。表 19.2 按优先级从高到低列出可用的运算符。运算符可以自由地结合成常量或可再定位表达式。连接器允许使用可再定位的表达式, 这种表达式要等到连接时才可以求解。

表 19.2 ASXA 运算符

运算符	意 义	优先级
UNL	测试参数为空	8
^	指数	7
*, / , MOD	乘、除、取模	6
SHR, SHL	右移、左移	6
ROR, ROL	右循环、左循环	6
+ -	一目加减、二目加减	5
HIGH	WORD 表达式的高字节	5
LOW	WORD 表达式的低字节	5
SEG	地址的段	5
EQ, NE, GT, GE, LT, LE	关系运算符	4
=, <, >, >=, >, <=, <	关系运算符	4
NOT	按位取反	3
AND	按位取与	2
OR	按位或	1
XOR	按位异或	1

### 19.2.12 汇编语句的格式

汇编语句的格式:

[标识符:] 助记符操作码 [操作数] [;注释]

其中:

- (1) 标识符:——标识符后跟冒号为标号, 代表语句的地址。
- (2) 助记符操作码——汇编指令操作码部分。
- (3) 操作数——汇编指令操作数部分。
- (4) 注释——语句的注释部分, 不产生执行代码。注释必须以分号为前导。

## 19.3 XA 汇编伪指令

### 19.3.1 伪指令语句格式

伪指令语句格式:

[名字] 伪指令 [操作数] [;注释]

其中:

- (1) 名字——对 MACRO, SET, EQU 才存在。
- (2) 伪指令——见表 19.3。
- (3) 操作数——见后。
- (4) 注释——语句的注释部分。注释必须以分号为前导。

表 19.3 ASXA 伪指令

伪指令	意 义
PUBLIC	使符号能被其他模块访问
EXTRN	使能访问其他模块的符号
GLOBAL	既适合 PUBLIC 又适合 EXTRN
END	结束汇编
PSECT	声明或假定程序段
ORG	设置位置计数器
EQU	定义符号值
SET	再定义符号值
DB	定义字节
DW	定义字
DF	定义实数
DS	保留存储
IF	条件汇编
ELSE	汇编另一部分
ENDIF	结束条件汇编
SIGNAT	指定 16 位签字值与标号相关联。连接时核对其值

### 19.3.2 PUBLIC

声明本模块中可被其他模块访问的多个符号(要用逗号分隔),如:

```
PUBLIC X1, X2, X3
```

### 19.3.3 EXTRN

声明所列的多个符号(要用逗号分隔)在其他模块也可被访问,如:

```
EXTRN X1, X2, X3
```

### 19.3.4 GLOBAL

所声明的多个符号(要用逗号分隔),其中在本模块定义的可被其他模块访问,不在本模块定义的就是要访问其他模块的符号,如:

```
GLOBAL X1, X2, X3
```

### 19.3.5 END

是可用可不用的,要用时一定放在程序最后。

### 19.3.6 程序段(PSECT)

PSECT 用于声明或恢复一个程序段。它需要一个名字和用逗号分隔的多个标志作为参数。程序段一旦被声明,下次再恢复时,只用名字作为参数就可以了,标志不必再用。表 19.4 给出 PSECT 可用的标志。

表 19.4 PSECT 可用标志

标 志	意 义
ABS	绝对段
GLOBAL	全局段,也是缺省段
LOCAL	非全局段
OVRLD	与其他一些模块相覆盖
PURE	只读段
RELOC	由指定边界上开始的可再定位段
ALIGN	段中所有标号都对齐到指定的边界上
SIZE	段的最大长度
BIT	按位为单位分配的段
CLASS	指定段的类名
SPACE	给段指定地址空间

(1) ABS——绝对段,从 0 开始。与覆盖段联合才是真正的绝对段,不然还可能与别的段接续。

(2) GLOBAL——全局段。连接时,可以与其他模块的同名全局段联合。

(3) LOCAL——局部段。连接时,不可以与其他局部段模块联合,就是同名也不可以。

(4) OVRLD——覆盖段。每个覆盖段对不足的部分作出贡献。与绝对段联合才是真正的绝对段,这时段内的变量才是绝对的。

(5) PURE——只读段,指明本段运行时不会改变。连接时可被放入 ROM。

(6) RELOC——允许向某个边界对齐的可再定位段,如 ALIGN=2 为偶地址对齐。

(7) ALIGN——段中所有标号都对齐到指定的边界上。如 ALIGN=2 为偶地址对齐。

(8) SIZE——为段指定最大长度。如 SIZE=100H。

(9) BIT——按位寻址段。段内地址按位分配。

(10) CLASS——为段指定类型。连接时用局部段中自己的名字无法连接,须借助类型才可进行连接。

(11) SPACE——给段指定地址空间。连接时要检查是否存在覆盖。

[例 19.2]

```
PSECT    fred
PSECT    bill, size = 100h, global
PSECT    john, abs, ovrl, class = code
```

### 19.3.7 ORG

本伪指令用它的参数值改变当前的位置计数器值。此值应是绝对地址或访问本段时的入口地址。如:

ORG 100h

### 19.3.8 EQU 和 SET

·EQU——定义符号并赋初值,如:

X0 EQU 123H

将操作数 123H 赋给标识符 X0。操作数可以用寄存器的保留字,这时的标识符就是寄存器的别名。

·SET——作用和 EQU 一样,但可以再定义符号。

### 19.3.9 DB 和 DW

DB 和 DW 对存储器分别按字节和字进行初始化。DB 和 DW 的操作数都是表达式表。对于 DB 的表达式表可以包括字符串。

[例 19.3]

alabel: DB 1,2,3,4,'X',"a string",0

DW 23\*20,alabel,k0,'a'

### 19.3.10 DF

DF 将存储器按双字进行初始化以存放实型数。在存储器中按 IEEE 标准存放实型数,但高字节在前。

[例 19.4]

pi: DF 3.14159

DF 3.3,2e10,-44

### 19.3.11 DS

DS 仅保留存储器空间而不初始化。操作数为表达式,其值为保留字节数。

[例 19.5]

xlabel: DS 34

ylabel: DS 3\*4

### 19.3.12 IF ELSE EKSEIF ENDIF

格式:

IF 符号	或	IF 符号
...		...
ELSE		ELSEIF 符号 1
...		...
ENDI		FELSE
		...
		ENDIF

其中:

符号——符号应是绝对表达式,对第一格式:其值为非零,对 IF 块做汇编;其值为零,对 ELSE 块做汇编。对第二格式:其值为非零,对 IF 块做汇编;其值为零,再看符号 1。符号 1 为非零,对 ELSEIF 块做汇编;其值为零,对 ELSE 块做汇编。

[例 19.6]

```
IF some_symbol
    mov r1,[r0]
ELSE
    move r1,[r0+]
ENDIF
```



### 19.3.13 SIGNAT

指定 16 位签字值与标号相关联。连接时对其值要进行核对。目前用于混合语言编程的函数调用。如:

```
SIGNAT _fred, 8192
```

其中 8192 是与 -fred 相关联的签字值。

### 19.3.14 控制选项伪指令行

本伪指令是放在源文件中作为一行而存在的,为了区别于汇编行,伪指令行用 \$ 打头。表 19.5 列出了汇编器控制选项伪指令和它们的作用。

表 19.5 汇编器控制选项伪指令

控制选项	缩写	缺省值	意义
PAGELNGTH(n)	PL	PL(66)	指定列表长度
PAGEWIDTH(n)	PW	PW	指定列表宽度
XR/NOXREF	XREF/NOXR	(120)	指定生成交叉访问表
CO/NOCOND	COND/NOCO	NOXR	指定是否显示条件汇编码
EFFECT	EJ	CO	指定换页
GE/NOGEN	GEN/NOGE		指定是否宏展开
INCLUDE(pathname)	IC	NOGE	指定包含文件所在路径
LIST/NOLIST	LI/NOLI		指定是否输出列表
SAVE/RESORE	SA/RS	LI	保存或恢复 LI, CO, GE
TITLE(string)	TT		指定列表的标题

## 19.4 宏

表 19.6 列出了 ASXA 的汇编宏。

表 19.6 ASXA 汇编宏

汇编宏	意义
MACRO	宏定义
ENDM	宏定义结束

续表 19.6

汇编宏	意 义
LOCAL	定义局部标号, 形式为 ?? nnnn
REPT	指定将块重复使用的次数
IRP	按给定的替换参数的个数将块重复多次
IRPC	按给定的替换字符串每次一字符将块重复多次
EXITM	结束宏展开

### 19.4.1 MACRO ENDM

#### 1. 定义宏格式:

```

宏名      MACRO      形式参数表
          (... 宏体 ...)
          ENDM

```

其中:

- (1) 宏名——宏体的标识符。
- (2) 形式参数表——逗号分隔的多个参数。

#### 2. 调用宏格式:

```

宏名      实参数表

```

[例 19.7] 定义宏:

```

xch      MACRO      reg1, reg2
          mov        r0, r@reg1
          mov        r@reg1, reg2
          mov        r@reg2, r0
          ENDM

```

调用宏:

```

xch      3, 4

```

宏展开:

```

mov      r0, 3
mov      r3, 4
mov      r4, r0

```

其中的@符在宏体中将其两侧的操作数结合起来, 自己不复存在。操作数中的一个必须是宏的参数。宏体中可使用 nul 操作符以测试宏参数。宏体中可以使用注释, 注释用双分号开头。但是, 注释在宏展开中被删掉, 以节省缓冲器。

### 19.4.2 LOCAL

LOCAL 保证宏体内生成的标号是惟一的。在使用过它之后, 宏展开中生成的标号有 ?? nnnn 的形式。

[例 19.8]

定义宏:

```

copy      MACRO      src, dst, cnt
          LOCAL      loop
          mov     r0, src
          mov     r1, dst
          mov     r2, cnt
loop:     mov     a, [r0+]
          mov     [r1+], a
          inc     r0
          inc     r1
          djnz    r2, loop
          ENDM

```

宏调用:

```
copy #inbuf, #procbuf, 32
```

宏展开:

```

          mov     r0, #inbufsrc
          mov     r1, #procbuf
          mov     r2, 32
?? 0001:  mov     a, [r0+]
          mov     [r1+], a
          inc     r0
          inc     r1
          djnz    r2, ?? 0001

```

### 19.4.3 REPT

使用 REPT 相当临时定义一个宏, 以实现将块重复指定的次数。

[例 19.9]

```

          mov     r0, #zbuf
          clr     r11
          REPT    3
          mov     [r0], r11
          inc     r0
          ENDM

```

宏扩展为:

```

          mov     r0, #zbuf
          clr     r11
          mov     [r0], r11
          inc     r0
          mov     [r0], r11
          inc     r0
          mov     [r0], r11
          inc     r0

```

超星浏览器提醒您:  
使用本复制品  
请尊重相关知识产权!



#### 19.4.4 IRP

与 REPT 相仿, 但将块按给定的替换参数的多少重复多次。

[例 19.10]

```
IRP      arg, lab1, lab2, #23
        mov    [r0], arg
        inc    r0
```

ENDM

宏扩展为:

```
        mov    [r0], lab1 inc r0 mov [r0], lab2
        inc    r0
        mov    [r0], #23
        inc    r0
```

超星浏览器提醒您:  
使用本复制品  
请尊重相关知识产权!

#### 19.4.5 IRPC

与 REPT 相仿, 但将块按给定的替换字符串每次一字符重复多次。

[例 19.11]

```
IRPC      arg, ABC
```

```
LOCAL     qq
```

```
        cjne    a, #'arg', qq
        jmp     case_@arg
```

qq:

ENDM

宏扩展为:

```
        cjne    a, #'A', ?? 0000
        jmp     case_A
?? 0000:    cjne    a, #'B', ?? 0001
        jmp     case_B
?? 0001:    cjne    a, #'C', ?? 0002
        jmp     case_C
?? 0002:
```

## 19.5 XA 汇编命令行

### 19.5.1 XA 汇编命令行格式

ASXA 汇编器可以运行于 MS-DOS 和 UNIX 之上, 用法一样。汇编命令行格式如下:

```
asxa    [控制选项]    文件表
```

其中:

- (1) 控制选项——可有可无。表 19.7 列出了汇编器的控制选项。
- (2) 文件表——逗号分隔的汇编源文件, 至少一个。

表 19.7 汇编器

控制选项	缺省值	意 义
<code>-Q</code>	优化汇编	快速汇编
<code>-U</code>		无未定义符号信息
<code>-S</code>		无尺寸错误信息
<code>-X</code>		OBJ 文件中无局部符号
<code>-O</code>	源文件.OBJ	指定输出文件名
<code>-L</code>	无列表	生成列表文件
<code>-W</code>	80 或多或少 32	指定列表页宽
<code>-F</code>	66	指定列表页长
<code>-I</code>	不进行宏展开	宏扩展列表
<code>-C</code>	无交叉访问	生成交叉访问表
<code>-V</code>	无行号	生成带行号信息的文件

下节分别介绍汇编控制选项。

### 19.5.2 汇编选项

介绍汇编控制选项如下。

1. `-Q`——加本选项只做两遍汇编,优化很少,但是速度较快。无本选项则反复对 jmp 进行优化直到代码最少。
2. `-U`——未定义的符号,汇编器作为外部符号处理,不加本选项则当作错误发布,加本选项则抑制错误的发布。
3. `-S`——字节量的值过大,发出错误信息。加本选项则抑制错误的发布。
4. `-X`——使目标文件的符号表中不包括局部符号,以减少文件的长度。
5. `-O` 目标文件名——指定目标文件名。不然,输出到第一源文件主名加 .OBJ 的文件中。
6. `-L` 列表文件名——指定列表文件名。不然,输出到标准输出设备。
7. `-Wn`——指定列表文件的行宽。n 为行字符数,缺省值:终端、打印机为 80;文件为 132。
8. `-Fn`——指定列表文件的页长。n 为页行数,缺省值为 66。
9. `-I`——本选项强制否定全部 NOLIST 伪指令行的作用,而是将所有的宏都展开,条件汇编都列出。
10. `-C`——指定产生交叉访问表并放在第一源文件主名加 .CRF 的文件中。再经 CREF 处理才会产生带格式的交叉访问表。
11. `-V`——本选项使目标文件带有行号和文件名等调试信息。如果是经编译器产生的目标文件自然带有行号和文件名,即无须加本选项。

## 第二十章 HLINK 连接器



### 20.1 简介

HLINK 连接器能够将多源文件程序分别经编译或汇编产生的目标文件连接成一个完整的可执行程序。由于 HI-TECH 提供的编译驱动程序(通过 HPD 和命令行)能自动地调用 HLINK 连接器,并自动选用必要的和合适的连接控制选项,所以,多数情况下不需要单独使用 HLINK 连接器。应该特别指出,单独使用 HLINK 连接器并非易事,需要有广泛而坚实地关于编译器和连接器的认识才可以用好。直接使用后面一章 HPDXA(51XA 集成开发平台)是最轻松而方便地完成同样任务的方法。一定要直接使用连接器的时候,最好是先照抄一个使用连接器命令行的例子,然后加以修改。这样可以保证不会遗漏任何必须的内容。

最后还应该指出的是,HLINK 连接器是一个适合多种 CPU C 编译器的通用连接器。下面所介绍的关于 51XA 的内容未必能适合其他 CPU 的编译器。

下面先介绍有关连接与定位(或装载)的基本概念,然后介绍连接命令。

### 20.2 连接与定位(或装载)的基本概念

1. 可再定位——连接的最基本的任务是将多个可再定位的目标文件连接成一个完整的可再定位的目标文件。可再定位的意思是在目标文件中存在有足够的信息,当此文件定位在存储空间的任意位置时,都能访问到本文件和其他文件的程序和数据的地址。再定位有两种基本形式:即按名字再定位和某一程序子段(psect)的基地址再定位。

2. 程序子段(psect)——基本程序子段有三种:text psect, data psect 和 bss psect。text psect 放可执行代码;data psect 放带初值外部变量;bss psect 放无初值外部变量。不同的 CPU 可能有自己独特的 psect。

3. 局部子段(local psect)——多数子段都是全局性质的,即任何模块用同一名字都可以访问到同一个子段。但有的子段却是局部性质的,即这个子段的名字只能被本模块所访问,即使对其他模块用同样的名字也访问不到。局部子段只有在连接时用类别名(汇编码 CLASS=指定)访问。

4. 全局符号——全局符号是 C 中使用了 extern 而未使用 static 存储类说明符的变量。对于汇编语言是说明为 global 的符号。连接器负责对全局符号进行匹配。

5. 连接地址与装载(或定位)地址——连接器要处理两类地址:连接地址与装载(或定位)地址。一般来说,psect 的连接地址是运行时进行存取要用的地址。装载(或定位)地址是对于输出文件(BIN 或 HEX 文件)来说,其 psect 的起始地址。它与连接地址可以相同也可以不相同。对于 8086 微处理器,粗略说来,连接地址相当段的偏移地址。装载(或定位)地址就是段地址乘以 16。带初值的 data psect 装载时地址在 ROM 空间,运行时应复制到 RAM 空间,使

用的是连接地址;又对于按体使用的 text psect,是按物理的装载(或定位)地址装入,按虚拟地址(即连接地址)运行的。应该指出,连接地址与装载(或定位)地址的运用更多地依赖于具体的编译器和存储模式。



## 20.3 连接命令

连接命令行的格式如下:

HLINK 控制选项表 目标文件表  
或 HLINK @命令文件名  
或 HLINK <命令文件名

其中:

(1) 控制选项表——选用表 20.1 中的 0 项或多项。

(2) 目标文件表——选用 1 个或多个目标文件。

(3) 连接命令行要求全部控制选项表和目標文件表均放在一行之內,一个逻辑行放不下时,可在行末使用续行符'\ '。

[例 20.1]

```
HLINK -Z -Ox.obj -Mx.map -Ptext=0,data=0/,bss,nvram=bss/ \
x.obj y.obj z.obj
```

(4) 命令文件名——连接命令行的第二和第三种格式都用到命令文件名。命令文件是扩展名为 .LNK 的文件。命令文件应把第一种格式的全部参数部分(即命令行除去 HLINK 的部分)都放在其中的文件。第二与第三种格式的作用一样。

[例 20.2]

HLINK @x.LNK  
或 HLINK <x.LNK

表 20.1 连接器控制选项表

控制选项	作用
- 8	使用 8086 的段;偏移 地址形式
- Apsect = 低地址 - 高地址, ...	为各 psect 指定地址范围
- Cpsect = 类名	给全局 psect 指定类名
- C 基地址	给二进制输出文件指定基地址
- D 符号文件	生成老式符号文件
- E 错误文件名	将错误信息重定向到指定的错误文件
- F	生成只有符号表的 .obj 文件
- G 段选择符	指定如何计算段选择符
- H 符号文件	生成指定的符号文件
- I	忽略未定义符号
- J 夭折错误数	指定产生夭折的最大错误数
- L	保留 .obj 文件中的再定位项
- LM	保留 .obj 文件段中的再定位项
- N	将映像文件的符号表按地址顺序排序

续表 20.1

控制选项	作用
-M 映像文件名	连接后生成指定的映像文件
-O 输出文件名	生成指定的输出文件
-Ppsect = 地址或顺序, ...	为各 psect 指定地址或顺序
-Spsect = 最大地址	指定 psect 的最大地址
-U 符号	把指定的符号预先添入未定义符号表
-Vav 符号表名	生成 Avocet 格式的符号文件
-W 告警级别	指定告警级别(-10~+10)
-W 映像文件宽度	指定映像文件宽度(>10)
-X	从符号表中去除所有的局部符号
-Z	从符号表中去除由于库函数自动生成的局部符号

## 20.4 OBJTOHEX 实用程序

HLINK 连接器只能生成简单二进制的 .obj 文件。其他转换后的二进制文件都需要经过 OBJTOHEX 实用程序处理才能生成。

OBJTOHEX 实用程序的命令行格式如下：

OBJTOHEX 控制选项 [输入文件名] [输出文件名]

其中：

(1) 控制选项——见表 20.2 OBJTOHEX 实用程序控制选项。

(2) 输入文件名——由 HLINK 生成的 .obj 文件。缺省文件名为 1.obj。

(3) 输出文件名——指定输出文件名。缺省输出文件名为 1.bin 或多或 1.hex 由选项 -b 确定。

表 20.2 OBJTOHEX 控制选项表

控制选项	意义
-8	产生 CP/M86 输出文件
-A	产生 ATDOS 的 .ATX 输出文件
-B 对基的偏移量	产生指定偏移量的二进制文件, 缺省文件名为 1.obj
-C[ckfile]	从 ckfile 或缺省时的标准输入读取校验和规范 *
-D	产生 .COD 文件
-E	产生 MS-DOS 的 .EXE 输出文件
-F[填充值]	将未用存储单元用填充值填充之, 缺省填充值为 0FFH
-I	产生 Intel HEX 输出文件, 使用扩展的线性地址记录格式
-L	将再定位信息传入输出文件(如 .EXE 文件)
-M	产生 Motorola HEX 输出文件(如 s14, s28, s37 等)
-N	产生 Minix 输出文件

续表 20.2

控制选项	意 义
-Pstk	产生具有指定堆栈的 Astari ST 输出文件
-R	产生包括再定位信息的输出文件
-S 文件名	将符号文件复制到指定文件
-T	产生 Tektronix HEX 输出文件, -TE 产生扩展的 TekHEX 文件
-U	产生 COFF 输出文件
-UB	产生 UBROF 格式输出文件
-V	在输出文件中将 word 和 long word 反序
-x	产生 x.out 格式文件

\* 注:校验和规范如下:

地址 1-地址 2 校验和 1-校验和 2 [+ 偏移量]

其中:

(1) 地址 1-地址 2——求取校验和的范围,取闭区间。使用十六进制数,但不加任何后缀。

(2) 校验和 1-校验和 2——存放校验和的范围,取闭区间。使用十六进制数,但不加任何后缀。存放校验和时,低字节在前则校验和 1 应小于校验和 2;高字节在前则校验和 1 应大于校验和 2。如果是 8 位的校验和,则两个地址的值应一样。

(3) + 偏移量——是校验和的初值。使用十六进制数,但不加任何后缀。缺省值为 0。

[例 20.3]

0005-1FFF 3-4 +1FFF

从字节 5 到字节 1FFFH(包括范围两端)求校验和。设为 16 位校验和,单元 3 放低字节,单元 4 放高字节。校验和的初值为末单元的地址 1FFFH。用以来检查全 0 的 ROM 和插错位置的 ROM。校验和的初值还可以有其他可能的应用。



# 第二十一章 HPDXA 51XA 集成开发平台

## 21.1 安 装



XAC 是运行于 MS \_DOS, UNIX 和 XENIX 不同的操作系统下的 51XA 微控制器的交叉编译器。下面介绍不同操作系统下 XAC 的安装。

### 21.1.1 MS \_DOS 下的安装

#### 1. 安装环境。

① 80x86 或 pentium CPU。② 512K 以上常规内存。③ 1M 以上 XMS 内存。④ 至少软、硬盘各一个。⑤ 建议使用鼠标器。⑥ MS \_DOS V3.3 或 DRDOS V6.0 以上操作系统。

2. XAC 放在多张软盘上提供。首盘的 PACKING.LST 列有文件清单。安装前应使用 SET 命令检查环境变量。一定要有 TEMP 变量,并要有有效的目录名。

3. 在盘符下键入 INSTALL 开始安装。屏幕上有信息窗和单行的状态行提示当前在做什么。有时还可能弹出对话框或告警框。

4. 全部安装或选装——安装过程中需要回答这个问题,有经验的人可以挑用选装,不然,应挑用全部安装。其余,问什么答什么。

5. 系列号和安装钥匙——在手册的标题页反面有系列号和安装钥匙,应正确的输入。

6. 之后,开始解压缩和复制程序,根据提示,适当操作。

7. 安装完毕,有可能需要修改 AUTOEXEC.BAT 或 CONFIG.SYS。如果需要修改,会提示如何修改。

8. 最后,建议阅读 READ.ME。那里有最新的技术信息。

9. 为了使修改的 AUTOEXEC.BAT 或 CONFIG.SYS 能够起作用,应重新启动计算机。

10. 运行交叉编译器。XAC 编译器的驱动程序有两个版本: XAC.EXE(命令行版)和 HPDXA.EXE(集成版)。命令行版的用法,前面已经介绍。下一章开始介绍集成版编译器的用法。

### 21.1.2 UNIX 操作系统下的安装

UNIX 操作系统下,51XA 微控制器的交叉编译器放在 TAR 格式的软盘或磁带上提供。具体安装如下:

#### 1. 创建安装目录:

• mkdir /usr/hitek。

• cd /usr/hitek。

#### 2. 安装命令: tar xf /dev/install。

#### 3. 设置 PATH 环境变量: SET PATH=/usr/hitek/bin。



4. 对 c\_shell 用户加: setenv HTC\_XA /home/hiteck。

对 Bourne shell 用户加: setenv

HTC\_XA = /home/hiteck

export HTC\_XA

5. 运行交叉编译器: 只有命令行版, 没有 HPDXA 版。命令行运行格式同 MS\_DOS 下的编译器。

## 21.2 快速入门



下面快速运行一个简单的程序。它是下一章的入门基础。

### 21.2.1 简单程序示例

```
#include <stdio.h>
void main()
{
    printf("Hello, world! \n");
}
```

### 21.2.2 使用 HPDXA

使用目标: 编辑上述源程序并且编译之。

·启动 HPDXA——键入 HPDXA 并回车。如果 HPD 安装正确并且放在搜索路径上, 就应该出现一个三视窗屏幕。最上面的视窗是一行的菜单条, 最下面的视窗是一行的信息窗, 其余的大面积视窗是编辑窗。

·编辑源程序——观察光标, 应该已经存在于编辑窗的左上方。跟着光标输入源程序正文。发现键入错误, 可及时修改。

·起名并存盘——键入 ALT-S, 弹出对话框要求输入文件名。键入欲用文件名如 HELLO.C 并回车。所编辑的正文已用所给名存入硬盘的当前目录中。

·编译——键入 F3, 弹出对话框要求输入程序 and 数据的定位地址。最简单的做法就是使用缺省值, 即在所用应输入处均按回车。之后, HPDXA 开始编译。错误多到一定程度, 编译器自动停止, 并在屏幕底部弹出错误信息窗, 与此同时光标指在错误行上。改正错误后, 重新键入 F3, 再次编译之。如此循环, 直到错误完全消除。这时, 当前目录中已生成 HELLO.HEX 文件。

·退出 HPDXA——键入 ALT-Q。

### 21.2.3 使用 XAC 命令行

这是另外一种编译源程序方法。它比使用 HPDXA 要繁琐。

·编辑源程序——可以使用任意一种编辑器, 只要它能够生成简单的 ASCII 文件即可。譬如 DOS 的 EDIT。设已完成源程序的编辑并起名 HELLO.C。

·编译——在 DOS 催促符下键入如下命令行:

## XAC HELLO.C

出现提示,要求输入存储器地址。同前,只按回车,使用缺省值。出现错误,终止编译。需要设法记住所有错误,重新进入编辑器去修改错误。改完后,退到 DOS,重新键入编译命令行。如此反复,直到编译成功,自动打印出存储器使用清单,并在当前目录中生成 HELLO.HEX 文件。

应该指出,这里的反复过程比使用 HPDXA 要繁琐得多。

## 21.2.4 运行程序

运行一个程序有许多情况,如在交叉模拟环境下运行,在仿真器环境下运行和在目标机环境下运行等。最终,当然是必须要在目标机环境下运行,其他都是为了调试而采用的中间环节。一般来说,为了调试和最终在目标机上运行,除了上述的软件外,在硬件上需要在线仿真器和 EPROM 编程器。关于在线仿真器和 EPROM 编程器的使用已经超出本书的范围。

本例的程序只有一条关于 printf( ) 的函数调用语句。它是标准 I/O 库里的函数,按照缺省它通过串行口输出到标准输出设备上,20MHZ 下波特率为 38 400。

## 21.3 HPDXA 用户接口

HPD 是 HI-TECH 的 C 程序开发系统。HPDXA 是专门用于 51XA 16 位微控制器的 C 语言程序开发的集成环境。它能方便地实现 C 语言编程(包括多语言混合编程)的编辑、编译、调试、运行等的一条龙服务。它具有友好的正文窗口界面。下面分监视器模式属性选择和下拉式菜单的命令操作来介绍用户接口。

## 21.3.1 监视器模式属性选择

1. 监视器模式——HPDXA 需要 EGA 或 VGA 以上显示卡的支持。EGA 卡支持 25~43 行,VGA 卡支持 25 行或 28~50 行。使用选项/SCREEN:nn 进行监视器模式选择,其中 nn 为 25,28,43,50。所选行数将自动存入 HPDXA.INI 文件中。今后每次进入 HPDXA 将自动选中,退出时,恢复原 DOS 屏幕行数。

2. 颜色选择——HPDXA 有两套颜色系统:屏幕的前景后景和彩色正文的单词辨色。表 21.1 和表 21.2 是用于第一套系统的颜色代码和颜色属性。表 21.3 是用于第二套系统的单词颜色代码符。

表 21.1 颜色代码

代 码	颜 色
0	黑
1	蓝
2	绿
3	青
4	红

表 21.2 颜色属性

属 性	说 明
normal	正常正文色
bright	高亮正文色
inverse	反转正文色
frame	视窗边框色
title	视窗标题色

续表 21.1

代 码	颜 色
5	品红
6	棕
7	白
8	灰
9	亮蓝
10	亮绿
11	亮青
12	亮红
13	亮品红
14	黄
15	亮白

表 21.3 彩色正文单词颜色代码符

单词颜色代码符	说 明
button	视窗按键色
C_wspace	空白符,前景色反映光标
C_number	数字
C_alpha	变量,宏,函数名
C_punct	标点符号
C_keyword	C关键字,如 int, static 等
C-brce	开括号,闭括号
C_s_quote	单引号中正文
C_d_quote	双引号中正文
C_comment	传统 C 注释
C++_comment	C++ 注释
C_preprocessor	C 预处理器伪指令
Include_file	包含文件名
Error	编辑器查到的错误
Asm_code	在线汇编码
Asm_comment	汇编器注释

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

如果希望修改这两个系统的颜色和属性可以根据表 21.1~21.3 的内容重新设置 HPDXA.INI 的有关项。修改缺省设置应采取慎重态度,并注意表 21.1 中的 8~15 号颜色只能用于前景色。

### 21.3.2 菜单命令操作

1. 进入 HPDXA 主屏幕(编辑屏幕)——在 DOS 催促符下键入编译器集成版驱动程序,格式如下:

HPDXA [文件名]

其中:

文件名有多种给法:

(1) 扩展名为 .c——在当前目录中找所指定的文件装入,未找到则创建此文件并进入编辑屏。

(2) 扩展名为 .prj——在当前目录中找所指定的工程文件装入,未找到则创建此工程文件并要求输入创建此工程文件所需要的文件表。

(3) 扩展名无——在当前目录中先按 .prj 工程文件找,有则装入;无,再按 .c 文件找,有则装入;无,则创建“untitled.c”空文件并进入编辑屏。

(4) 文件名无——为缺省状态,这时创建名为“untitled.c”空文件并进入编辑屏。

2. 键盘与鼠标——菜单命令的操作除了使用键盘之外还可以使用鼠标或者二者混用。HPDXA 支持标准的 int 33h 接口的鼠标器驱动程序。老式的鼠标器驱动程序在 HPDXA 下也能工作,但是由于缺少一些状态调用,在系统菜单下进行鼠标器设置时将失败。

3. 菜单条——HPDXA 主屏幕的上部有一行菜单条。菜单条分成图标或文字的小块。每块的图标或文字代表一个下拉菜单的标题。

4. 菜单条的操作——表 21.4 列出了键盘和鼠标对菜单条的操作。

表 21.4 操作菜单条

操 作	键 盘	鼠 标
进入菜单条	Alt - space	左键点菜单条或中键点屏幕任意处
退出菜单条	Alt - space 或 Esc	左键点菜单系统之外
后个菜单的标题	→	在菜单条上按左键右拖到目标标题放开
前个菜单的标题	←	在菜单条上按左键左拖到目标标题放开

5. 下拉菜单命令项的操作——表 21.5 列出了键盘和鼠标对下拉菜单命令项的操作。

表 21.5 操作下拉菜单的命令项

操 作	键 盘	鼠 标
选中命令项	1. 用 ↓ ↑ 键点亮命令项, 再按回车键 2. 使用命令项对应的热键(见表 8.6)	1. 在欲选命令项上点左键或中键 2. 由菜单条连拖下来在欲选命令项上放开

表 21.6 下拉菜单命令项对应的热键

下拉菜单	命令项	对应热键
打开下拉菜单	File	Alt - F
	Edit	Alt - E
	Compile	Alt - I
	Make	Alt - M
	Run	Alt - R
	Utility	Alt - U
	Help	Alt - H
File	Open	Alt - O
	New	Alt - N
	Save	Alt - S
	Save as	Alt - A
	Quit	Alt - Q
Edit	Cut	Alt - X
	Copy	Alt - C
	Paste	Alt - V
Compile	Warning level	Alt - W
	Optimization	Alt - Z
	Compile and link	F3
	Compile to OBJ	Shift - F3
	Compile to AS	Ctrl - F3

续表 21.6

下拉菜单	命令项	对应热键
Make	Load project ...	Alt - P
	Make	F5
	Re - link	Shift - F5
	Re - make	Ctrl - F5
Run	DOS command ...	Alt - D
	DOS shell	Alt - J
Utility	User command 1	Shift - F7
	User command 2	Shift - F8
	User command 3	Shift - F9
	User command 4	Shift - F10
Editor	search	F2

超星阅读器提醒您：  
使用本复制品  
请尊重相关知识产权！

6. 系统下拉菜单——菜单条最右端的图标<<>>是系统下拉菜单的标题符号。打开系统下拉菜单,有两个命令项:Setup 和 About。执行 About 命令将给出 HPDXA 的版本信息。执行 Setup 将给出 DOS 系统的设置和存储器使用情况,还有关于鼠标器的灵敏度、时钟和声音报警等设置。

7. 视窗的操作——HPDXA 的多视窗可以前后覆盖和平铺,所以有一个视窗选中的问题。HPDXA 的多数视窗还可以移动或缩放。下面表 21.7~表 21.8 介绍视窗的这些操作。

表 21.7 HPDXA 视窗选择


工 具	操 作	要 点
键 盘	重复使用 Ctrl - 回车	每次将最后窗循环到最前
鼠标器	1. 重复使用 Alt - 左键点当前正文区 2. 左键点非当前但可见窗的正文区	每次将最前窗循环到最后 直接选中

注:有的窗一旦出现就不动,除非关闭,如错误信息窗。这种窗只要能看见就可以使用。

表 21.8 HPDXA 视窗移动/缩放

目 的	工 具	操 作	要 点
移 动	键 盘	<ul style="list-style-type: none"> <li>·按 Ctrl - Alt - space 进入移动/移动模式</li> <li>·按←→↑↓进行移动</li> <li>·按回车退出本模式,或其后菜单系统的任意操作自动迫其退出</li> </ul>	
	鼠标器	<ul style="list-style-type: none"> <li>·在视窗的任意角上用鼠标找出菱形光标,按左键拖之,可向任意方向移动</li> <li>·在视窗的上、下边框的中部用鼠标找出垂直箭头光标,按左键拖之,可向垂直方向移动</li> <li>·在视窗的左、右边框的中部用鼠标找出水平箭头光标,按左键拖之,可向水平方向移动</li> </ul>	<ul style="list-style-type: none"> <li>·按正确方法找不出变形光标的就是不可移动的视窗</li> <li>·所有可见视窗都可试用本操作</li> </ul>

续表 21.8

目 的	工 具	操 作	要 点
缩 放	键 盘	<ul style="list-style-type: none"> <li>·按 Ctrl + Alt + space 进入移动/移动模式</li> <li>·按 Shift 加←进行缩,按 Shift 加→进行放</li> <li>·按回车退出本模式,或其后菜单系统的任意操作自动迫其退出</li> </ul>	
	鼠标器	<ul style="list-style-type: none"> <li>·在视窗的任意角上用鼠标找出菱形光标,按右键拖之,可向任意方向缩放</li> <li>·在视窗的上、下边框的中部用鼠标找出垂直箭头光标,按右键拖之,可向垂直方向缩放</li> <li>·在视窗的左、右边框的中部用鼠标找出水平箭头光标,按右键拖之,可向水平方向缩放</li> </ul>	 <ul style="list-style-type: none"> <li>·按正确方法找不出变形光标的就是不可缩放的视窗</li> <li>·所有可见视窗都可试用本操作</li> </ul>

8. 按钮的操作——有一些视窗自己带有按钮的标识,在这种按钮上照例都有标有等价的键符,可以当作热键,用键盘输入;也可以用鼠标器左键点此按钮。

## 21.4 HPDCA 菜单命令快览

本节对 HPDCA 的各个菜单命令作简单的介绍。介绍时,按菜单条上的子菜单标题自左至右展开进行。

### 21.4.1 系统子菜单(<<>>)

1. Setup …——鼠标器的灵敏度、时钟和声音报警等的设置,DOS 版本号,CPU 型号,显示器模式和存储器使用情况等。
2. About …——HPDCA 的版本号和许可证信息等。

### 21.4.2 File 子菜单

包括管理文件的命令和退出 HPDCA 的命令。

1. Open …——为编辑器加载已存在的文件。
2. New——为编辑器以“untitled”为名创建新文件。
3. Save——保存已存在的文件。如果是“untitled”为名的文件则要求起名。
4. Save as …——要求为欲保存文件起名。
5. Autosave …——弹出对话框要求输入以分钟为单位的时间间隔。它是自动保存到临时文件的间隔时间。当 HPDCA 非正常退出时,为下次进入文件时提供手动恢复的条件。
6. Quit——退入 DOS。

### 21.4.3 Edit 子菜单

包括一般剪切板、查找替换正文等编辑命令外,还有针对 C 的缩进缩出、添减注释、彩色正文选色等命令。



1. Cut——将选定内容切除放在剪切板上, 只要没有新的内容进入剪切板, 就可多次用于粘贴。
2. Copy——将选定内容复制到剪切板上, 只要没有新的内容进入剪切板, 就可多次用于粘贴。
3. Paste——将剪切板上最新进入的内容, 多次地粘贴到光标处。
4. Hide——这是 Wordstar 风格的命令, 用于将选中的块区隐去或显示。
5. Delete selection——将选定内容直接删除, 并不进入剪切板。
6. Search ...——搜索字符串。
7. Replace ...——替换字符串。
8. Show clipboard——显示或关闭剪切板视窗。视窗可被编辑。
9. Go to line ...——直接找到指定行的正文。
10. Set tab size ...——指定制表符的步长, 范围 1~16, 缺省为 8。
11. Indent——将选定的块缩进一制表符的步长。
12. Outdent——将选定的块伸出一制表符的步长。
13. Comment/Uncomment——将选定块的每行开始插入或删除 C++ 风格的注释符 (//)。插入或删除决定于原来的状态。
14. C colour coding——彩色与非彩色正文的切换。正文为彩色时菜单命令前显示出标志。



#### 21.4.4 Option 子菜单

本子菜单为编译器设置选项, 对于工程文件也起作用。

1. Memory model and chip type ...——指定芯片型号、存储模式等。
2. Output file type——指定输出文件类型, 缺省为 Intel HEX 文件。
3. ROM & RAM addresses ...——指定 ROM, RAM, nvram 等的地址。
4. ROM checksum specs ...——指定校验和的判据规范。可取 1~4 字节的校验和, 缺省为 2。
5. Long formats in printf( )——本选项指示 HLINK 链入 long 版本的 printf( ), 即格式串中含有 %ld, %lu, %lx 样的参数。本版本将增加代码的长度。仅仅是使用长整型数进行计算, 并不需要本选项。选中本选项, 菜单命令前显示出标志。
6. Float formats in printf( )——本选项指示 HLINK 链入 float 版本的 printf( ), 即格式串中含有 %e, %f, %g 样的参数。本版本将增加代码的长度。仅仅是使用浮点数进行计算, 并不需要本选项。选中本选项, 菜单命令前显示出标志。
7. Source level debug info——指定在当前符号文件中是否包含源级调试信息。在仿真器上使用 HI-TECH 的 LUCIFER 调试软件时需要这个信息。
8. Suppress local symbols——指示在符号文件中删除局部符号。
9. Avocet format symbol file——指示生成与 AVSIM 兼容的 Avocet 符号文件。

#### 21.4.5 Compile 子菜单

本子菜单主要是关于单文件的编译/连接。



1. Compile and link——本命令对单一源文件首先进行编译,接着自动运行连接器和实用程序 objto.EXE 以产生可执行的 .EXE 文件。如对 HELLO.C 使用本命令产生 HELLO.EXE。如果所给源文件的扩展名为 .AS,会自动改用汇编器并最后产生可执行的 .EXE 文件。

2. Compile to .OBJ——只编译到生成 .OBJ 文件。

3. Compile to .AS——只编译到生成 .AS 文件。

4. Stop on Warnings——指定非灾难性错误是否停止编译。本命令为切换性命令,有效时菜单命令上出现标志。

5. Warning level ...——报警级别,当前可用级别为 -9~9,缺省为 0,值越小越挑剔。

6. Optimization ...——选择优化级别。对工程文件同样有效。

7. Identifier length ...——选择标识符长度,范围为 31~255,缺省为 31。

8. Pre-process assembler files——汇编源文件预处理。使汇编源文件与 C 源文件一样,能在汇编前进行宏展开和确定条件汇编。选定后,菜单命令上出现标志。

9. Generate assembler listing——指定对于汇编源文件和 C 源文件都产生扩展名为 .LST 的汇编列表文件。

10. Generate C source listings——指定生成包括源代码、行号、制表符的 C 列表文件。文件扩展名仍为 .LST。

#### 21.4.6 Make 子菜单

本子菜单专门统筹高效处理多源文件程序的开发,直到已调试通过的可执行代码。实现的步骤如下:

- 创建工程文件。
- 输入工程用源文件表。
- 建立特定库表、预定义预处理器符号表、目标文件表、连接选项表等。
- 保存工程文件。
- 用实用程序 Make 或 Remake 生成可执行代码。

1. Make——能分辨源文件的类型,自动选用汇编器或编译器进行编译生成 .OBJ 文件,能选出自上次编译后又被修改过的源文件重新编译,能发现自上次连接后又有新的 .OBJ 文件出现须再次连接以产生新的输出文件,能感知所使用的头文件中有的已被修改,应将所有依赖的源文件都重新编译等。

2. Re-make——相当于废弃原来的 .OBJ 文件和以它们为基础的全部文件,强迫从编译全部源文件开始重做 Make。

3. Re-link——重新对当前的工程做连接。丢失和未被更新的 .OBJ 文件都会得到应有地处理。

4. Load project ...——装入已存在的工程。

5. New project ...——创建新工程。弹出对话框要求输入一系列的配置参数,如芯片型号、存储类型、输出文件类型、存储地址等,还有工程用源文件表。

6. Save project——将当前工程存入文件。

7. Re-name project ...——给工程改名。

8. Output file name ...——要求指定工程输出文件名,缺省名为工程名加扩展名 .EXE。

9. Map file name ...——确定是否为当前工程生成映像文件。选定要生成映像文件,则在



号文件。LUCIFER 只能下载 Intel HEX, Motorola HEX 和 BIN 类型的输出文件。

4. Debugger——本命令使用当前符号文件运行 LUCIFER 调试软件。本命令并未下载用户代码,所以,可以被用做返回到 LUCIFER 挂起状态。

5. Debugger setup ...——执行本命令弹出对话框要求为 LUCIFER 调试软件设置串口参数。参数一旦设置,将和其他设置一样存入工程文件。对于 AX 版的缺省 LUCIFER 设置参数已放在 luax\_args 中。

6. Auto download after compile——本命令用于打开或关闭自动下装。打开时,菜单命令上出现标志。这时,编译一条龙成功的完成,由 HPDXA 启动 LUCIFER 下载输出文件并加载符号文件。

#### 21.4.8 Utility 子菜单

本子菜单提供一些有用的实用程序。

1. String search——本实用程序功能较强,能一次设置搜索多个文件或当前工程中的文件。

2. Memory usage map——执行本命令将显示最近一次存储空间使用表。用 ESC 键关闭本显示窗,或用鼠标器左键点左上角关闭图标。

3. Calculator——这是一个程序员使用的软件台式多视窗计算器。能进行二、八、十、十六等进制的整数运算,结果用四种进制数同时显示。操作和真的计算器一样,只是按键。也可以用鼠标器左键点按屏上的软键。计算器视窗可以移动。关闭使用 ESC 键,或用鼠标器左键点左上角关闭图标或计算器右下角 OFF 软键。

4. Ascii table——这是一个 ASCII 符的查表窗,上面有四个按钮,可以查看四种进制的字符值。关闭查表窗与前项相仿。

5. Define user commands ...——用户可以定义四个 DOS 形式的命令。定义时使用表 21.9 所列的预定义宏。使用本命令弹出的对话框来定义这四个用户命令。每个用户命令都有热键,shift-F7~shift-F10 分别用于用户命令 1~4。

表 21.9 定义用户命令使用的宏

类 型	宏 名	宏体意义
用 做 路 径	\$(LIB)	系统标准库文件所在目录,如 C:\HPDXA\LIB
	\$(CWD)	当前工作目录
	\$(INC)	系统标准头文件所在的 INCLUDE 目录
用 做 文 件 名	\$(EDIT)	当前编辑器中装载的文件名。此文件被修改时,宏体被替换为用于自动保存的临时文件名,文件卸载时又替换为本文件名。这种机制能使外部编辑器对临时文件的修改容易地反映到 HPDXA 的编辑器中来
	\$(OUTFILE)	即可执行文件名
	\$(PROJ)	当前工程的主名(如 AUDIO.PRJ 的 AUDIO)

### 21.4.9 Help 子菜单

Help 子菜单给你许多方面的帮助。HPDXA 启动时在当前目录和 HELP 目录中寻找 TBL 文件,它们应当加到 Help 子菜单中。HELP 目录的路径由环境变量 HT-xx-HLP 设置。如果没有设置,在 HPDXA 执行时会由全路径中求出。如果 HELP 目录找不到,所有标准 HELP 项都用不起来。

1. HPDXA——本命令列出关于 HPDXA 的帮助项目。
2. C Library Reference——本命令给出标准 ANSI C 库的在线手册。
3. XA Compiler Options——本命令列出编译器的控制选项。
4. Release notes——本命令给出本版发行备注。

## 21.5 HPDXA 编辑器

HPDXA 具有内附的专门服务于 C 语言开发一条龙的编辑器。它除去具有一般编辑器应有的功能外,还有一些服务于预处理器、编译、连接(/定位)等自动排除明确错误的功能。下面仅介绍一般编辑器的功能,其他的特殊功能在下一节或具体开发过程中才会遇到。

内附编辑器是在 WordStar 的基础上增加 Ms \_ DOS/Microsoft Windows 编辑器的风格和操作形成的。熟悉 Ms \_ DOS/Microsoft Windows 或 WordStar 的人都可以很快上手。对初学者建议沿着 Windows 菜单风格和操作的方向走下去,键盘与鼠标操作并重。下面仅扼要指出 Edit 子菜单中特别一些的命令。

1. Hide, Indent, Outdent——它们是 WordStar 风格的操作命令。Hide 是将选中的块在显示和隐藏之间切换。Indent 将选中的块缩进一个制表符步长。Outdent 相反,将选中的块伸出一个表符步长。
2. Show clipboard, Clear clipboard, Delete selection——它们是 Winsows 风格的 clipboard 操作命令。它们直接放到子菜单中,对于增强 clipboard 的编辑灵活性很有好处。
3. Go to line..., Comment/Uncomment——是直接 C 语言有关的命令,对排错和修改源文件很有好处。详见 HPDXA 菜单命令一节。
4. C colour coding, Set tab size——是关于相关内容设置的,见 HPDXA 菜单命令一节。

## 21.6 编译连接一条龙示例

HPDXA 提供两套编译连接一条龙服务。单文件程序用 Compile 子菜单命令,多文件程序用 Make 子菜单命令。本节用单文件程序的编译连接一条龙示例作为入门。

1. 建议首先建立工作目录:

```
c: \ > md mywork  
c: \ > cd mywork
```

2. 进入集成开发平台并编辑源文件:

```
c: \ mywork > HPDXA
```

- (1) 出现一个三视窗屏幕。最上面的视窗是一行的菜单条,最下面的视窗是一行的信息

窗,其余的大面积视窗是编辑窗。

(2) 编辑源程序——观察光标,应该已经存在于编辑窗的左上方。跟着光标输入源程序正文。发现键入错误,可及时修改。源程序正文如下:

```
#include <stdio.h>
void main( )
{
    printf("Hello, world! \n")
}
```

这个程序就是快速入门一节的程序,只不过有意去掉了 `printf()` 语句最后的结束符 `;`, 目的在于观察编辑器的报错反映。

(3) 存盘——有三种方法都可以作存盘操作:①键盘:先进入主菜单条用 `alt-space`, 找 File 用 `→`, 再找 Save 用 `↓` 键,最后用回车键。②鼠标器:左键点主菜单条 File 并下拖到 Save 松键。③用热键 `ALT-S`。这时,弹出对话框要求输入文件名。键入欲用文件名 `HELLO.C` 并回车。所编辑的正文已用所给名存入硬盘的当前目录中。

(4) 设编译选项——主要是设置存储地址。操作:①键盘:先进入主菜单条用 `alt-space`, 找 Option 用 `→`, 再找 ROM RAM Addresses... 用 `↓` 键,最后用回车键。②鼠标器:左键点主菜单条 Option 并下拖到 ROM RAM Addresses... 松键。这时,弹出对话框要求输入各种地址。填 ROM—0, RAM—20, RAM size—1e0, NVRAM—0。

(5) 编译并连接——用 Compile 子菜单的 Compile and link 命令。操作:①键盘:先进入主菜单条用 `alt-space`, 找 Compile 用 `→`, 再找 Compile and link 用 `↓` 键,最后用回车键。②鼠标器:左键点主菜单条 Compile 并下拖到 Compile and link 松键。③用热键 `F3`。开始编译并连接。

(6) 错误信息窗——由于埋伏了一个错误,编译或连接停止,并在屏幕底部弹出错误信息窗。与此同时,光标指在错误行上。本例由于错误行没有结束符,只能指在下行的开始。错误信息窗的正文指出发生的错误和错误发生在哪个阶段。多数错误发生在编译一趟(P1.EXE),代码生成(CGxx.EXE)预处理器(CPP.EXE)阶段,连接阶段(LINK.EXE)也会产生错误。错误信息窗的上边框是关于错误的统计信息,本例的错误信息正文是:

; expected

错误信息窗的下部有许多按钮,这些按钮是因情况而变的。本例中,因错误很明确,便有 FIX 按钮出现。按下此按钮会自动修改错误。如果没有 FIX 按钮出现,就意味着无法自动修改。这时就得人工修改错误。如果是警告,会有 CONTINUE 按钮出现。按下此按钮会容忍警告而继续向下执行。如果是有多错误,会一个又一个地提请自动或是人工修改。另外,错误信息窗还可以用 HIDE 命令隐去或显示。

(7) 再次编译并连接——再次编译并连接时,将省去存盘而直接使用 Compile and link 命令。HPDXA 将修改过的文件自动存入暂时文件,随即编译并连接。如此循环直到编译并连接成功,并存盘下装,经调试通过,完成开发。

# 附录

## 附录 A C51 函数库

### A.1 数学函数

本节函数的原型说明在头文件 math.h 中。

#### A.1.1 函数名: abs, cabs, fabs, labs

原型:

```
extern int abs (int val);
extern char cabs (char val);
extern float fabs (float val);
extern long labs(long val);
```

功能:

abs 求变量 val 的绝对值, 即 val 为正原样返回, 为负则返回其负数。这四个函数除了变量和返回值的数据类型不同外, 功能是相同的。

[例 A.1]

```
test_xabs( )
#include <math.h>
#include <stdio.h> /* for printf */
void main(void)
{ int x1, y1;
  char x2, y2;
  float x3, y3;

  x1 = -12;          y1 = abs(x1);
  x2 = -23;          y2 = cabs(x2);
  x3 = -3.4;         y3 = fabs(x3);
  x4 = -12345L;      y4 = labs(x4);

  printf("abs( %d ) = %d \n", x1, y1);
  printf("cabs( %bd ) = %bd \n", x2, y2);
  printf("fabs( %f ) = %f \n", x3, y3);
  printf("labs( %ld ) = %ld \n", x4, y4);
```

超星浏览器提醒您:  
使用本复制品  
请尊重相关知识产权!



### A.1.2 函数名:exp, log, log10

原型:

```
extern float exp(float x);
extern float log(float x);
extern float log10(float x);
```

功能:

exp 返回以 e 为底的  $x$  次幂; log 返回  $x$  的自然对数( $e=2.718\ 282$ ); log10 返回  $x$  的以 10 为底的对数。

[例 A.2]

```
test_explog( )
#include <math.h>
#include <stdio.h>    /* for printf */
void      main(void)

{ float  x1, y1;
  float  x2, y2;
  float  x3, y3;

      x1=4.60517;          y1=exp(x1);
      x2=y1;              y2=log(x2);
      x3=10000;           y3=log10(x3);

      printf("exp( %f ) = %f \n", x1, y1);
      printf("log( %f ) = %f \n", x2, y2);
      printf("log10( %f ) = %f \n", x3, y3);
}
```

### A.1.3 函数名:sqrt

原型:

```
extern float sqrt(float x);
```

功能:

sqrt 返回  $x$  的正平方根。

[例 A.3]

```
test_sqrt( )
#include <math.h>
#include <stdio.h>    /* for printf */
void      main(void)
{
    float  x, y;
```



```
x=16.0;
y=sqrt(x);
printf("sqrt(%f)=%f\n", x, y);
```



#### A.1.4 函数名:rand,srand

原型:

```
extern int rand();
extern void srand(int n);
```

功能:

rand 返回一个由 0~32 767 的伪随机数。srand 用来将随机数发生器初始化成一个已知 (或期望) 的值, 使 rand 的后继调用产生相同序列的随机数。

[例 A.4]

```
test_xrand()
#include <math.h>
#include <stdio.h> /* for printf */
void main(void)
{
    int x;

    for(x=0; x<10; x++)
        printf("x=%d, rand = %d\n", x, rand());

    srand(32);
    for(x=0; x<10; x++)
        printf("x=%d, rand = %d\n", x, rand());
}
```

#### A.1.5 函数名:cos,sin,tan

原型:

```
extern float cos(float x);
extern float sin(float x);
extern float tan(float x);
```

功能:

cos 返回  $x$  的余弦值; sin 返回  $x$  的正弦值; tan 返回  $x$  的正切值。所有函数变量范围为  $-\pi/2 \sim +\pi/2$ , 变量必须在  $\pm 65\,535$  之间, 否则会产生 NAN 错误。

[例 A.5]

```
test_SinCosTan()
#include <math.h>
#include <stdio.h> /* for printf */
```

```

void      main(void)
{
    float   x;
    float   y1, y2;

    for(x=0.0; x<3.141 6; x += 0.1)
    {
        y1=sin(x);    printf("sin( %f )= %f \n", x, y1);
        y2=cos(x);    printf("COS( %f )= %f \n", x, y2);
        printf("tan( %f )= %f \n", x, tan(x));
    }
}

```



### A.1.6 函数名: acos, asin, atan, atan2

原型:

```

extern float  acos (float x);
extern float  asin (float x);
extern float  atan (float x);
extern float  atan (float y, float x);

```

功能:

acos 返回  $x$  的反余弦值; asin 返回  $x$  的反正弦值; atan 返回  $x$  的反正切值。它们的值域为  $-\pi/2 \sim +\pi/2$ 。atan2 返回  $x/y$  的反正切, 其值域为  $-\pi \sim +\pi$ 。

[例 A.6]

```

test_arcSinCosTan( )
#include <math.h>
#include <stdio.h>    /* for printf */
void      main(void)
{
    float   x;
    float   y1, y2, y3;

    for(x=0.0; x<3.141 6; x += 0.1)
    {
        y1=asin(x);    printf( "asin( %f )= %f \n", x, y1);
        y2=acos(x);    printf("ACOS( %f )= %f \n", x, y2);
        printf("atan( %f )= %f \n", x, atan(x));
        printf("atan2( %f )= %f \n", x, atan2(x));
    }
}

```

### A.1.7 函数名: cosh, sinh, tanh

原型:

```

extern float  cosh (float x);

```

```
extern float sinh (float x);
extern float tanh (float x);
```

功能:

cosh 返回  $x$  的双曲余弦值;sinh 返回  $x$  的双曲正弦值;tanh 返回  $x$  的双曲正切值。

[例 A.7]

```
test _ SinhCoshTanh( )
#include <math.h>
#include <stdio.h> /* for printf */
void main(void)
{ float x;
  float y1,y2;

  for(x=0.0; x<3.141 6; x += 0.1)
  {
    y1=sinh(x); printf( "sinh( %f )= %f \n", x, y1);
    y2=cosh(x); printf( "COSH( %f )= %f \n", x, y2);
    printf( "tanh( %f )= %f \n", x, tanh(x));
  }
}
```

超星阅读器提醒您：  
使用本复制品  
请尊重相关知识产权！

### A.1.8 函数名:fpsave, fprestore

原型:

```
extern void fpsave (struct FPBUF * P);
extern void fprestore (struct FPBUF * P);
```

功能:

fpsave 保存浮点程序的状态。fprestore 将浮点程序的状态恢复为其原始状态,当用中断程序执行浮点运算时这两个函数是有用的。

[例 A.8]

```
test _ floatpoint( )
#include <math.h>
#include <stdio.h> /* for printf */
void main(void)
{ struct FPBUF fp_save_buf;
  float x,y,z;

  fpsave(&fp_save_buf);
  x=1.3;
  y=3.141 6;
  z=( sin(x) + tan(x))/ y ;

  printf( "z= %f \n", z);
}
```

```
fprestore(&fp_save_buf);
```

### A.1.9 函数名:ceil

原型:

```
extern float ceil(float x);
```

功能:

返回不小于  $x$  的最小整数(仍是浮点数返回)。

[例 A.9]

```
test_ceil()  
#include <math.h>  
#include <stdio.h>    /* for printf */  
void      main(void)  
{  
    float  x, y;  
  
    x = 12.3;  
    y = ceil(x);    /* y gets 13.0 */  
    printf("ceil( %f ) = %f \n", x, y);  
}
```

### A.1.10 函数名:floor

原型:

```
extern float floor(float x);
```

功能:

返回不大于  $x$  的最大整数(仍是浮点数返回)。

[例 A.10]

```
test_floor()  
#include <math.h>  
#include <stdio.h>    /* for printf */  
void      main(void)  
{  
    float  x, y;  
  
    x = 3.9;  
    y = floor(x);    /* y gets 3.0 */  
    printf("floor( %f ) = %f \n", x, y);  
}
```

### A.1.11 函数名:modf

原型:



```
extern float modf (float x, float *ip);
```

功能:

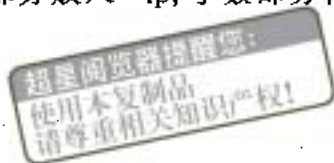
将  $x$  分为整数和小数部分,二者都有  $x$  的相同符号,整数部分放入  $*ip$ ,小数部分作为返回值。

[例 A.11]

```
test _ modf( )
#include <math.h>
#include <stdio.h> /* for printf */
void      main(void)
{
    float  x, integer_part, fraction_part;

    x = 234.56

    fraction_part = modf( x, & integer_part);
    printf( "%f + %f * %f \n", x, integer_part, fraction_part );
}
```



### A.1.12 函数名: pow

原型:

```
extern float pow (float x, float y);
```

功能:

求  $xy$  值并返回之。由于所给参数不合理,致结果为“非数”,返回 NAN。当  $x=0$  而  $y$  为负实数或  $x$  为负实数而  $y$  不是整数时,返回出错码。

[例 A.12]

```
test _ pow( )
#include <math.h>
#include <stdio.h> /* for printf */
void      main(void)
{
    float  x, base, power;

    base = 2.0;
    power = 8.0;
    x = pow(base, power);
    printf( "%f = %f ^ %f \n", x, base, power );
}
```

## A.2 标准化 I/O 函数

本节函数为字符 I/O 函数。它们通过串行口操作。为支持其他 I/O 设备,应改写 getkey

( ) 和 putchar( ) 函数。为保持 getkey( ) 和 putchar( ) 正常工作, 串行口应事先初始化。

### A.2.1 函数名: \_getkey( )

原型:

```
extern char _getkey( );
```

功能:

\_getkey( ) 从 8051 串行口读入一个字符, 然后等待下个字符输入。这个函数是和输入设备接口的惟一函数, 输入设备改变时它应作相应的修改。

[例 A.13]

```
test _getkey( )
#include <stdio.h>
void      main(void)
{ char    c;

    while( c=_getkey( ) != 0x1B)
    {
        printf( "key= %c  %bu,  %bx \n", c, c, c );
    }
}
```

### A.2.2 函数名: getchar

原型:

```
extern char getchar( );
```

功能:

getchar 使用 \_getkey 从串行口读入字符, 除了读入的字符马上传给 putchar 函数以作响应外, 与 \_getkey 相同。

[例 A.14]

```
test _getchar( )
#include <stdio.h>
void      main(void)
{ char    c;

    while( c=getchar( ) != 0x1B)
    {
        printf( "character= %c  %bu,  %bx \n", c, c, c );
    }
}
```

### A.2.3 函数名: gets

原型:

```
extern char *gets( char * s, int n);
```



功能:

本函数通过 `getchar` 由输入设备读入字符串送字符数组。考虑到 ANSI 标准的建议, 限制每次调用时读入的最大字符数。本函数提供了一个字符计数器“n”。在读入字符时, 检测到换行符停止读字符, 并送 \0 提前结束字符串的输入。

[例 A.15]

```
test_gets( )
#include <stdio.h>
void      main(void)
{ xdata  char  buf[50];

        gets( buf, sizeof( buf ) );
        printf( "Input string is \" %s \"\n", buf );
}
```



#### A.2.4 函数名: `ungetchar`

原型:

```
extern char ungetchar (char);
```

功能:

`ungetchar` 将输入字符推回输入缓冲区, 供下次 `gets` 或 `getchar` 使用。`ungetchar` 成功时返回 `char`, 失败时返回 `EOF`。

[例 A.16]

```
test_ungetchar( )
#include <stdio.h>
void      main(void)
{ char    c;
  int      i=0;

        puts("input an integer followed by a char: ");
        while(isdigit( c = getchar( )))
            i = 10 * i + c - 0x30 ;
        if( c! = EOF)
            ungetchar( c );
        printf("\n i = %d, and next char in buffer = %c.\n", i, c );
}
```

#### A.2.5 函数名: `_ungetkey`

原型:

```
extern char _ungetkey (char);
```

功能:

`_ungetkey` 将输入的字符送回输入缓冲区, 并将其值返回给调用者。下次使用 `_getkey` 仍可获得该字符。



### A.2.6 函数名: putchar

原型:

```
extern putchar (char);
```

功能:

putchar 通过 8051 串口输出一字符。和函数 \_getkey 一样, putchar 是与输出设备惟一接口函数, 当输出设备改变时, 它是惟一需要修改的。

[例 A.17]

```
test_putchar( )
#include <stdio.h>
```

```
void      main(void)
{ unsigned char  i;

    for( i=0x20; i<0x7f; i++ )
        putchar(i);
}
```

### A.2.7 函数名: printf

原型:

```
extern int printf (const char *, ...);
```

功能:

printf 以第一参数指向的格式字符串指定的格式从 8051 串口输出字符串和变量值, 返回实际输出的字符数。

注:

printf 参数的总字节数 SMALL 和 COMPACT 模式不超过 15 个字节, LARGE 模式下, 至多 40 个字节。格式字符串中变量格式部分包含下列各域(方括号内的域是可选的):

```
%[flags][width][.precision] type
```

其中:

(1) [flags]——标志, 可有可无。

- 输出结果左对齐。
- + 输出符号数时前面加 + / - 号。
- “ 输出 + / - 号时 + 号以空格代替。
- # 与 0, x(X) 连用, 输出的数字以 0, 0x(0X) 为前导。
- 与 f, g(G), e(E) 连用, 输出的结果中一定包括小数点。
- b, B 与 d, i, u, o, x(X) 连用, 变量改为 8 位数, 如 %bu。
- l, L 与 d, i, u, o, x(X) 连用, 变量改为 32 位数, 如 %lu。
- \* 抑制下一个变量不输出。

(2) [width]——指定欲显示的字符串最小宽度, 可有可无。

- n 十进制正整数。如果实际字符数小于 n, 左端补空格。

0n	如果实际字符数小于 $n$ , 左端补 0。
*	在参数表中指定宽度。
(3) [.precision]——精度指定符, 可有可无。	
(无)	对 d, i, u, o, x(X) 为 1。 对 f, e(E) 为 6。 对 g(G) 为所有有效数字。 对 s, c 为一字符。
.0	对 d, i, u, o, x(X) 精度同缺省。 对 f, g(G), e(E) 不输出小数部分。
.n	精度要求输出 $n$ 个十进制位或 $n$ 个字符, 不足左添零。 超出舍入或丢弃。
.*	在参数表中指定精度。
(4) type——变量类型。	
d, i	int(十进制符号数)。
u	unsigned int(十进制无符号数)。
o	int(八进制无符号数)。
x, X	int(十六进制无符号数)。x 时, 用 abcdef; X 时, 用 ABCDEF。
f	float([ - ]ddd.ddd 的符号数)。
e, E	float([ - ]d.ddd e[ + / - ]ddd 的符号数, [ - ]d.ddd E[ + / - ]ddd 的符号数)。
g, G	float(在 f, e 中选最适合给定值的形式)。
c	char(单个字符)。
s	string pointer(ASCII 字符串)。
p	pointer(C:aaaa, D:aaaa, I:aaaa, P:aaaa 其中:C—CODE, D—DATA, I—IDATA, P—PDATA, aaaa—指针偏移量)。

## [例 A.18]

```

test _ printf( )
#include <stdio.h>
void      main(void)
{ char  c;
  int    i;
  long   l;
  unsigned char  uc;
  unsigned int   ui;
  unsigned long  ul;
  float  x, y;
  char  buf[ ] = "test string";
  char  *p = buf;


```

```

c = 2;

```





```

i = 2345;
l = 0x7ffffff;

uc = 'B';
ui = 12 345;
ul = 0x3456abcd;

f = 20.0;
g = 44.98;

printf("char %bd, int %d, long %ld \n", c, i, l);
printf("unsigned char %bu, unsigned int %u, unsigned long %lu \n", uc, ui, ul);
printf("The same as above, but in hexadecimal number: %bx, %x, %lx \n", uc, ui, ul);

printf("%f and %f in exponential format: %e and %e \n", x, y, x, y);
printf("%f and %f in more decision and compact form: %g and %g \n", x, y, x, y);

printf(" The address of string %s is at %p. \n", buf, p);

```

### A.2.8 函数名:sprintf

原型:

```
extern int sprintf(char s, const char *, ...);
```

功能:

sprintf 与 printf 相似,但输出不显示在控制台上,而是输出到指针指向的缓冲区。

注:

sprintf 允许参数字节数与 printf 相同。

[例 A.19]

```

test _ sprintf( )
#include <stdio.h>
void main(void)
{ char buf[50];
  int n, x, y;
  float f;

  x = 222;
  y = 345;
  f = 3.141 59;

  n = sprintf( buf, "%f \n", 1.2 );
  n += sprintf( buf + n, "%d \n", x );
  n += sprintf( buf + n, "%d %s %g \n", y, " - - -", f);
}

```

```
printf(buf);
```



### A.2.9 函数名: puts

原型:

```
extern int puts (const char *s)
```

功能:

puts 将字符串 s 和换行符写入控制台设备。执行错误时返回 EOF, 正确时返回字符数。

[例 A.20]

```
test_puts( )
#include <stdio.h>
void main(void)
{
    puts("put this string out using putchar as a line.");
}
```

### A.2.10 函数名: scanf

原型:

```
extern int scanf(const char *, ...);
```

功能:

scanf 在作为第一个参数的格式字符串控制下, 利用 getchar 函数由控制台读入的字符序列转换成指定的数据类型, 按顺序赋予对应的指针变量。scanf 返回转换过的项数, 出错则返回 EOF。

格式字符串中的格式部分包含下列各域(方括号内的域是可选的):

%[flags][width] type

其中:

(1) [flags]——标志, 可有可无。内容:

b, h	与 d, i, u, o, x(X)连用, 指针改为字符指针, 如 %bu。
l	与 d, i, u, o, x(X)连用, 指针改为长指针, 如 %lu。
*	抑制本输入项。

(2) [width]——指定由控制台输入最多字符数。在读入给定字符数之前遇到空白符或不可转换字符时, 即行停止。

n 十进制正整数。如果实际字符数小于 n, 左端补空格。

(3) type——将读入的字符数转换成的变量类型。

d, i	int(十进制符号数)。
u	unsigned int(十进制无符号数)。
o	int(八进制无符号数)。
x, X	int(十六进制无符号数)。
f, e, E, g, G	float(浮点数)。

c	char(单个字符)。
s	string pointera(ASCII 字符串)。



格式串中的格式部分用空格符或制表符分隔。格式部分的数目,应与参数表中除第一个参数外的地址变量相匹配,严格禁止输入地址变量数偏少,否则会引发灾难性后果。

从输入设备上输入变量的 ASCII 字符串时,要按每个格式部分的 WIDTH 字段的规定输入,每个格式部分输入字符数可以小于各自 WIDTH 字段规定的字符数,具体实现是提前用空格键或 TAB 键结束本字段,随后即可接着输入下个变量的 ASCII 字符串,直到最后以回车键结束全部输入。如果输入某个变量的 ASCII 串字符数超出了规定长度,内部会自动丢弃超出的部分。

本函数输入实参时的总字符数受 8051 内存的限制,即 small 和 compact 模式为 15 个字符;large 模式为 40 个字符。

[例 A.21]

```
test _ scanf( )
#include <stdio.h>
void main(void)
{ char c;
  int i;
  long l;
  unsigned char uc;
  unsigned int ui;
  unsigned long ul;
  float x, y;
  char buf[10];
  int n;

  printf("Enter signed char, int, and long numbers respectively, please:");
  n = scanf("%bd %d %ld", c, i, l);
  printf("There %d numbers are read. \n", n);

  printf("Enter unsigned char, int, and long numbers respectively, please:");
  n = scanf("%bu %u %lu", uc, ui, lu);
  printf("There %d numbers are read. \n", n);

  printf("Enter a char, and a string respectively, please:");
  n = scanf("%c %9s", c, buf);
  printf("There %d arguments are read. \n", n);

  printf("Enter two floating-point numbers respectively, please:");
  n = scanf("%f %8f", x, y);
  printf("There %d numbers are read. \n", n);
}
```

### A.2.11 函数名:sscanf

原型:

```
extern int sscanf(char *s, const char *, ...);
```

功能:

sscanf 与 scanf 方式相似,但是输入不是通过控制台,而是由以 '\0' 结尾的字符串输入。

注:

sscanf 输入的实参总字节数受 8051 内存的限制,在 small 和 compact 模式时最大允许 15 字节;在 large 模式下,最大可允许 40 个字节。

[例 A.22]

```
test _sscanf( )
#include <stdio.h>
void      main(void)
{ char  c;
  int   i;
  long  l;
  unsigned char  uc;
  unsigned int   ui;
  unsigned long  ul;
  float  x, y;
  char  buf[10 ];
  int   n;

  printf("Enter signed char , int , and long numbers respectively, please:");
  n=sscanf("2   -345  678 912", "%bd  %d  %ld", c, i, l );
  printf("There %d numbers are read. \n", n );

  printf("Enter unsigned char , int , and long numbers respectively, please:");
  n=sscanf("3   44  56 781 234", "%bu  %u  %lu", uc, ui, lu );
  printf("There %d numbers are read. \n", n );

  printf("Enter a char , and a string respectively, please:");
  n=sscanf("a  blacksmith", "%c  %9s", c, buf );
  printf("There %d arguments are read. \n", n );

  printf("Enter two floating - point numbers respectively, please:");
  n=sscanf("3.141 59  22.6", "%f  %8f", x, y );
  printf("There %d numbers are read. \n", n );
}
```

## A.3 动态存储函数

本节函数的原型在头文件 stdlib.h 中。



### A.3.1 函数名: calloc

原型:

```
void *calloc(unsigned int n, unsigned int size);
```

功能:

calloc 分配  $n$  个  $size$  大小的堆中内存块, 并将该块的首地址返回。分配未成功, 则返回 NULL 指针。分配的内存块被初始化为 0。

[例 A.23]

```
test_calloc( )
#include <stdlib.h>
#include <stdio.h> /* for printf */
void main(void)
{ int xdata *p;

  p = calloc( 100, sizeof(int));
  if (p = NULL)
    printf("Error in allocating memory array \n");
  else
    printf("The allocated array address is %p\n", (void *)p);
}
```

### A.3.2 函数名: free

原型:

```
void free(void xdata *p);
```

功能:

释放指针  $p$  所指向的内存块, 指针清为 NULL。

[例 A.24]

```
test_free( )
#include <stdlib.h>
#include <stdio.h> /* for printf */
void main(void)
{ void *mbuf;

  printf("Allocating memory \n");
  mbuf = malloc( 100);
  if (mbuf = NULL)
    printf("Unable to allocate memory array \n");
  else
    free(mbuf);
  printf("Memory has been freed");
}
```





### A.3.3 函数名: init\_mempool

原型:

```
void init_mempool(void xdata *p, unsigned int size);
```

功能:

初始化动态分配管理的堆。p 为首址指针, size 为管理区大小。

[例 A.25]

```
test _init_mempool( )
#include <stdlib.h>
void      main(void)
{
    void xdata  *p;
    int    i;

    init_mempool(&XBYTE[0x2000], 0x1000 );
    p = malloc(1000);
    for(i=0; i<1000; i++)
        ((char *)p)[i] = i & 0xFF;
    free(p);
}
```

### A.3.4 函数名: malloc

原型:

```
void * malloc(unsigned int size);
```

功能:

从堆中动态分配 size 大小的存储块,并返回该块的首址指针。分配不成功返回 NULL 指针。分配块不初始化。

[例 A.26]

```
test _malloc( )
#include <stdlib.h>
#include <stdio.h>

void      main(void)
{
    unsigned char xdata  *p;
    int    i;

    p = malloc(100);
    if (p = NULL)
        printf("Error in allocating memory array \n" );
    else
        printf("The allocated array address is %p \n", (void *)p );
}
```



### A.3.5 函数名:realloc

原型:

```
void *realloc(unsigned xdata *p, unsigned int size);
```

功能:

改变 p 所指 d 的分配块的大小,原分配块内容复制到新块中,新块较大时,多余部分也不作初始化。返回新块首址指针,如果新块未分配成功,则保持原块不动,并返回 NULL 指针。

# [例 A.27]

```
test _ realloc( )
#include <stdlib.h>
#include <stdio.h>

void main(void)
{
    void xdata *p;

    p = malloc(100);
    .
    .
    .
    if (p != NULL)
    {
        p = realloc(p, 180);
        if (p != NULL)
            printf("Reallocate successfully.");
        else
            printf("Failed in reallocating \n");
    }
    else
        printf("Failed in allocating \n");
}
```

## A.4 字符归类函数

本节函数原型说明在头文件 ctype.h 中。

### A.4.1 函数名:isalpha

原型:

```
extern bit isalpha(char);
```

功能:

isalpha 检查传入的字符是否在 A~Z 和 a~z 之间,真返回为 1,否则为 0。

[例 A.28]



```

test _isalpha( )
#include <ctype.h>
#include <stdio.h>    /* for printf */
void      main(void)
{ unsigned char  c;
  char          *p;

  for( c=0; c<128; c++ )
  { p=( isalpha(c) ? "yes" : "no" );
    printf( "%c is an alphabetic character ? %s! \n", c, p );
  }
}

```



#### A.4.2 函数名: isalnum

原型:

```
extern bit isalnum(char);
```

功能:

isalnum 检查变量是否位于 A~Z 和 a~z 或 0~9 之间, 真返回 1, 否则为 0。

[例 A.29]

```

test _isalnum( )
#include <ctype.h>
#include <stdio.h>    /* for printf */
void      main(void)
{ unsigned char  c;
  char          *p;

  for( c=0; c<128; c++ )
  { p=( isalnum(c) ? "yes" : "no" );
    printf( "%c is an alphanumeric character ? %s! \n", c, p );
  }
}

```

#### A.4.3 函数名: iscntrl

原型:

```
extern bit iscntrl(char);
```

功能:

iscntrl 检查变量值是否在 0x00~0x1F 之间或等于 0x7F, 真返回 1, 否则为 0。

[例 A.30]

```

test _iscntrl( )
#include <ctype.h>
#include <stdio.h>    /* for printf */

```

```

void      main(void)
{
    unsigned char  c;
    char           *p;

    for( c=0; c<128; c++ )
    {
        p=( iscntrl(c) ? "yes" : "no" );
        printf( "%c is a control character ? %s! \n", c, p );
    }
}

```



#### A.4.4 函数名:isdigit

原型:

```
extern bit isdigit(char);
```

功能:

isdigit 检查变量值是否在 0~9 之间,真返回 1,否则为 0。

[例 A.31]

```

test _ isdigit( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void      main(void)
{
    unsigned char  c;
    char           *p;

    for( c=0; c<128; c++ )
    {
        p=( isdigit(c) ? "yes" : "no" );
        printf( "%c is a decimal digit ? %s! \n", c, p );
    }
}

```

#### A.4.5 函数名:isgraph

原型:

```
extern bit isgraph(char);
```

功能:

isgraph 检查变量是否为可打印字符,可打印字符的值域为 0x21~0x7E。若可打印,返回 1,否则为 0。

[例 A.32]

```

test _ isgraph( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void      main(void)
{
    unsigned char  c;

```

```
char      *p;
```

```
for( c=0; c<128; c++ )
{ p=( isgraph(c) ? "yes" : "no" );
  printf( "%c is a printable character (not including space)? %s! \n", c, p );
}
```

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

#### A.4.6 函数名:isprint

原型:

```
extern bit isprint(char);
```

功能:

除与 isgraph 相同外,还接受空格符(0x20)。

[例 A.33]

```
test_isprint( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void main(void)
{ unsigned char c;
  char *p;

  for( c=0; c<128; c++ )
  { p=( isprint(c) ? "yes" : "no" );
    printf( "%c is a printable character ( including space)? %s! \n", c, p );
  }
}
```

#### A.4.7 函数名:ispunct

原型:

```
extern bit ispunct(char);
```

功能:

ispunct 检查字符变量是否为 ASCII 字符集中的标点符号或空格。如果是其中之一,则返回 1, 否则返回 0。标点符号包括 iscntrl( )全部字符外加下列字符:

! # \$ % ^ & \* ( ) - + = \ | } [ ] : " ? ' < > , . ? /

[例 A.34]

```
test_ispunct( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void main(void)
{ unsigned char c;
  char *p;
```

```

for( c=0; c<128; c++ )
{
    p=( ispunct(c) ? "yes" : "no" );
    printf( "%c is a punctuation character ? %s! \n", c, p );
}

```



#### A.4.8 函数名: islower

原型:

```
extern bit islower(char);
```

功能:

islower 检查字符变量值是否位于  $a \sim z$  之间, 真返回 1, 否则返回 0。

[例 A.35]

```

test_islower( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void main(void)
{
    unsigned char c;
    char *p;

    for( c=0; c<128; c++ )
    {
        p=( islower(c) ? "yes": "no" );
        printf( "%c is a lowercase alphabetic character ? %s! \n", c, p );
    }
}

```

#### A.4.9 函数名: isupper

原型:

```
extern bit isupper(char);
```

功能:

isupper 检查字符变量是否位于  $A \sim Z$  之间, 真返回 1, 否则返回 0。

[例 A.36]

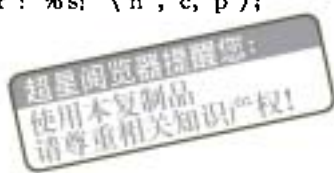
```

test_isupper( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void main(void)
{
    unsigned char c;
    char *p;

    for( c=0; c<128; c++ )
    {
        p=( isupper(c) ? "yes" : "no" );
    }
}

```

```
printf( "%c is an uppercase alphabetic character ? %s! \n", c, p );
```



#### A.4.10 函数名: isspace

原型:

```
extern bit isspace(char);
```

功能:

isspace 检查字符变量是否为下列之一: 空格、制表符、回车、换行、垂直制表符和送纸符。如果真返回 1, 否则返回为 0。

[例 A.37]

```
test_isspace( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void main(void)
{ unsigned char c;
  char *p;

  for( c=0; c<128; c++ )
  { p=( isspace(c) ? "yes" : "no" );
    printf( "%c is a whitespace character ? %s! \n", c, p );
  }
}
```

#### A.4.11 函数名: isxdigit

原型:

```
extern bit isxdigit(char);
```

功能:

isxdigit 检查字符变量是否位于 0~9, A~F 及 a~f 之间, 是返回 1, 否则为 0。

[例 A.38]

```
test_isxdigit( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void main(void)
{ unsigned char c;
  char *p;

  for( c=0; c<128; c++ )
  { p=( isxdigit(c) ? "yes" : "no" );
    printf( "%c is a hexadecimal digit ? %s! \n", c, p );
  }
}
```



**A.4.12 函数名: toascii(参数宏)**

原型:

```
toascii( c ) (( c ) & 0x7f )
```

功能:

用参数宏将任何整型值的低 7 位取出构成有效的 ASCII 字符。

[例 A.39]

```
test _ toascii( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void main(void)
{ unsigned char k;
  int c=0xff41;

  k=toascii( c ); /* k=0x41 or 'A' */
  printf( "%d is converted to ascii character %c. \n", c, k );
}
```

**A.4.13 函数名: toint**

原型:

```
extern char toint(char);
```

功能:

toint 将十六进制数对应的 ASCII 字符(0~9 或 A~F 或 a~f)转换为整型数 0~15, 并返回该整型数。ASCII 字符不在上述范围的不作转换, 直接返回原字符。

[例 A.40]

```
test _ toint( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void main(void)
{ unsigned long n;
  char k;

  for( n=0; isdigit( k=getchar( ) ); n *= 10)
    n += toint( k );
  printf( "The input number in ascii is converted to long int. as %lx. \n", n );
}
```

**A.4.14 函数名: tolower**

原型:

```
extern char tolower(char);
```



功能:

tolower 将字符转换为小写字符。如果字符不在 A~Z 之间,则不作转换,直接返回原字符。

[例 A.41]

```
test _tolower( )
#include <ctype.h>
#include <stdio.h>    /* for printf */
void      main(void)
{ unsigned char k;

    for( k=0x20; k< 0x7f; k++ )
        printf( "tolower( %bu ) = %c. \n", k, tolower(k) );
}
```



#### A.4.15 函数名:\_tolower(参数宏)

原型:

```
tolower ( c ) ( c - 'A' + 'a' )
```

功能:

该宏相当于参数值加 0x20。

[例 A.42]

```
test _tolower( )
#include <ctype.h>
#include <stdio.h>    /* for printf */
void      main(void)
{ char k;

    puts("pease input a character in upper case: ");

    if (isupper( k=getchar( ) ))
    { printf( " the input character is %c. \n", k);
      k = _tolower( k );
      printf( " the input character is %c. \n", k);
    }
}
```

#### A.4.16 函数名:toupper

原型:

```
extern char toupper (char);
```

功能:

toupper 将字符变量转换为大写字符。如果字符变量不在 a~z 之间,则不作转换,返回原字符。

## [例 A.43]

```
test _toupper( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void main(void)
{ char *string = "This is a string. ";
  int k, length;
  for( k=0; k < length = strlen( string ); k++ )
    string[k] = toupper(string[k] );
  string[k] = '\0';
  printf("Now, the string is: %s. \n", string );
}
```

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

## A.4.17 函数名：\_toupper(参数宏)

原型：

```
_toupper( c )    ( ( c) - 'a' + 'A' )
```

功能：

该宏相当于参数值减 0x20。

## [例 A.44]

```
test __toupper( )
#include <ctype.h>
#include <stdio.h> /* for printf */
void main(void)
{ char j, k;

  j=0;
  for( k=0x20; k<0xff; k++ )
  { if(islower(k))
    buf[j++] = toupper( k );
  }
  buf[j] = '\0';
  printf( "The characters in buf are: %s. \n", buf );
}
```

## A.5 字符串函数

本节函数的原型说明放在 string.h 头文件中。在函数 memcmp, memcpy, memchr, memccpy, memmove 和 memset 中,字符串长度由调用者用参数明确规定,这些函数可工作在任何模式下。

## A.5.1 函数名:memchr

原型：

```
extern void *memchr(void *s1, char val, int len);
```

功能:

memchr 在串  $s_1$  的前 len 个字符中找出字符 val。成功时返回  $s_1$  中的第一个指向 val 的指针, 失败时返回 NULL。

[例 A.45]

```
test _ memchr( )
#include <string.h>
#include <stdio.h> /* for printf */
void main(void)
{ static char buf[100] = "This is the string intended to be searched.";
  void *p;

  p = memchr( buf, 'g', strlen(buf) );
  if (p != NULL)
    printf("'g' is found in the buffer at position %d. \n", p - buf);
  else
    printf("'g' is not found in the buffer. ");
}
```

超星浏览器提醒您:  
使用本复制品  
请尊重相关知识产权!

### A.5.2 函数名: memcmp

原型:

```
extern char memcmp(void *s1, void *s2, int len);
```

功能:

memcmp 逐个字符比较串  $s_1$  和串  $s_2$  的前 len 个字符。相等时返回 0, 如果串  $s_1$  大于或小于  $s_2$ , 则相应返回正数或负数。

[例 A.46]

```
test _ memcmp( )
#include <string.h>
#include <stdio.h> /* for printf */
void main(void)
{ static char buf[100] = "This is the STRING intended to COMPAR.";
  static char buf1[100] = "This is the string intended to be compared.";
  char i;

  i = memcmp( buf, buf1, strlen(buf) );
  if (i == 0)
    printf("The string in buf equals to the string in buf1.");
  else if (i > 0)
    printf("The string in buf is bigger than the string in buf1.");
  else
    printf("The string in buf is smaller than the string in buf1.");
}
```



### A.5.3 函数名: memcpy

原型: `extern void *memcpy(void *dest, void *src, int len);`

功能:

memcpy 由 src 所指向的内存中拷贝 len 个字符到 dest 中。返回指向 dest 最后一个字符的指针。如果 src 和 dest 相交迭, 结果不可预测。

[例 A.47]

```
test_memcpy( )
#include <string.h>
#include <stdio.h> /* for printf */
void main(void)
{ static char buf[100] = "This is the STRING intended to be COPIED.";
  static char buf1[100] = "0123456789abcdef";
  char *p;

  printf("Before memcpy( ), the buf contains string: %s. \n", buf );
  p = memcpy( buf, buf1, strlen(buf) );
  if (p)
    printf("After memcpy( ), the buf contains string: %s. \n", buf );
  else
    printf("Failed. memcpy( ) can't guarantee the source of being overlapped \n");
}
```

### A.5.4 函数名: memccpy

原型:

`extern void *memccpy(void *dest, void *src, char val, int len);`

功能:

memccpy 将 src 前 len 个字符拷贝到 dest 中。拷贝完 len 个字符返回 NULL。中途遇字符 val 则拷贝后停止, 并返回指向 dest 中的下个元素的指针。

[例 A.48]

```
test_memccpy( )
#include <string.h>
#include <stdio.h> /* for printf */
void main(void)
{ static char buf[100] = "This is the STRING intended to be COPIED.";
  static char buf1[100] = "0123456789abcdef";
  char *p;

  printf("Before memccpy( ), the buf contains string: %s. \n", buf );
  p = (char *) memccpy( buf, buf1, 'c', strlen(buf1) );
```

```
if (p)
{ *p= '\0';
printf("The character array is: %s\n", s);
```



### A.5.7 函数名: strcat

原型:

```
extern char *strcat(char *s1, char *s2);
```

功能:

strcat 将串 s<sub>2</sub> 拷贝到串 s<sub>1</sub> 末尾。假定 s<sub>1</sub> 串足以容纳两个串。返回指向 s<sub>1</sub> 串的第一个字符的指针。

[例 A.51]

```
test_strcat( )

#include <string.h>
#include <stdio.h> /* for printf */

int main(void)
{ char buf[100];
  char *blank = " ", *c = "c + +", *visual = "Visual";
  strcpy( buf, visual );
  strcat(buf, blank );
  strcat(buf, c );
  printf("%s. \n", buf );

  return(0);
}
```

### A.5.8 函数名: strncat

原型:

```
extern char *strncat(char *s1, char *s2, int n);
```

功能:

strncat 拷贝串 s<sub>2</sub> 中 n 个字符到 s<sub>1</sub> 末尾, 如果 s<sub>2</sub> 比 n 短, 则只拷贝 s<sub>2</sub> (包括串结束符)。

[例 A.52]

```
test_strncat( )

#include <string.h>
#include <stdio.h> /* for printf */

int main(void)
{ char buf[30];
  int n;
  strcpy( buf, "class No. " );
  n = strlen(buf);
  strncat(buf, "fivtten", sizeof(buf) - n);
  printf("%s. \n", buf );

  return(0);
}
```





### A.5.9 函数名: strcmp

原型:

```
extern char strcmp(char *s1, char *s2);
```

功能:

strcmp 比较串  $s_1$  和  $s_2$ 。如果相等返回 0。如果  $s_1 < s_2$ , 返回负数,  $s_1 > s_2$ , 返回正数。

[例 A.53]

```
test _ strcmp( )
#include <string.h>
#include <stdio.h> /* for printf */
int main(void)
{ char *buf1 = "aaa", *buf2 = "bbb";
  int i;

  i = strcmp( buf1, buf2 );
  if( ! i )
    printf("buf1 equals buf2. \n ");
  elseif( i > 0 )
    printf("buf1 is greater than buf2. \n ");
  else
    printf("buf1 is smaller than buf2. \n ");

  return(0);
```

### A.5.10 函数名: strncmp

原型:

```
extern char strncmp(char *s1, char *s2, int n);
```

功能:

strncmp 比较串  $s_1$  和  $s_2$  中前  $n$  个字符。返回值与 strcmp 相同。

[例 A.54]

```
test _ strncmp( )
#include <string.h>
#include <stdio.h> /* for printf */
int main(void)
{ char *buf1 = "aaa", *buf2 = "bbb";
  int i, n;

  n = 2;
  i = strncmp( buf1, buf2, n );
  if( ! i )
```



```

    printf("As comparing the first %d characters , buf1 equals buf2. \n");
elseif( i > 0 )
    printf("As comparing the first %d characters , buf1 is greater than buf2. \n");
else
    printf("As comparing the first %d characters , buf1 is smaller than buf2. \n");

return(0);

```



### A.5.11 函数名: strcpy

原型:

```
extern char *strcpy(char *s1, char *s2);
```

功能:

strcpy 将串 s<sub>2</sub>, 包括结束符, 拷贝到 s<sub>1</sub>。返回指向 s<sub>1</sub> 的第一个字符的指针。

[例 A.55]

```

test _ strcpy( )
#include <string.h>
#include <stdio.h> /* for printf */
int main(void)
{ char *buf1 = "aaa", *buf2 = "bbb";
  int i;

  printf("buf1: %s. \n ", buf1);
  i = strcpy( buf1, buf2 );
  printf("buf1: %s. \n ", buf1);

  return(0);
}

```

### A.5.12 函数名: strncpy

原型:

```
extern char *strncpy(char *s1, char *s2, int n);
```

功能:

strncpy 与 strcpy 相似, 但只拷贝前 n 个字符, 如果 s<sub>2</sub> 长度小于 n, 则 s<sub>1</sub> 串以 0 补齐到长度 n。

[例 A.56]

```

test _ strncpy( )
#include <string.h>
#include <stdio.h> /* for printf */
int main(void)
{ char *buf1 = "aaa", *buf2 = "bbb";

```

```

int    i;

printf("buf1: %s. \n ", buf1);
i = strncpy( buf1, buf2, 2 );
buf[3] = '\0';
printf("buf1: %s. \n ", buf1);
return(0);
}

```

### A.5.13 函数名:strlen

原型:

```
extern int strlen(char *s1);
```

功能:

strlen 返回串 s<sub>1</sub> 中字符个数(不包括串结束符)。

[例 A.57]

```

test_strlen( )
#include <string.h>
#include <stdio.h> /* for printf */
int    main(void)
{   char    *buf = "aaabbbcccd";

    printf("The length of string is %d. \n ", strlen( buf ) );

    return(0);
}

```

### A.5.14 函数名:strchr, strpos

原型:

```
extern char *strchr(char *s1, char c);
```

```
extern int strpos(char *s1, char c);
```

功能:

strchr 搜索 s<sub>1</sub> 串中第一个出现的 'c' 字符。如果成功, 返回指向该字符的指针。搜索也包括结束符。因此搜索一个空字符串时, 返回指向串结束符的指针, 不是空指针。

strpos 与 strchr 相似, 但它返回字符在串中的位置。s<sub>1</sub> 串的第一个字符位置是 0。失败返回为 -1。

[例 A.58]

```

test_strchr_strpos( )
#include <string.h>
#include <stdio.h> /* for printf */
int    main(void)
{   char    *buf = "aaabbbcccd";

```



```

char    *p;
int     n;

p = strchr(buf, 'b');
if ( p )
{
    printf("The character %c is found at the position: %d. \n", c, p - buf );
    n = strpos( buf, 'b' );
    if( n == -1 )
        printf("The character was not found. \n");
    else
        printf("The character position got by strpos( ): %d. \n", n );
}
else
    printf("The character wasn't found. \n");

return(0);

```



### A.5.15 函数名: strrchr, strrpos

原型:

```

extern char *strrchr ( char *s1, char c );
extern int  *strrpos ( char *s1, char c );

```

功能:

strrchr 搜索串  $s_1$  中最后一次出现的字符 'c'。成功返回指向该字符的指针, 否则返回 NULL。  $s_1$  为空串时, 返回指向串结束符的指针, 而不是空指针。

strrpos 与 strrchr 相似, 但它返回字符在串中的位置, 失败返回为 -1。

[例 A.59]

```

test _strrchr _strrpos( )
#include <string.h>
#include <stdio.h>    /* for printf */
int     main(void)
{   char    *buf = "aaabbbcccd";
    char    *p;
    int     n;

    p = strrchr(buf, 'b');
    if ( p )
    {
        n = strrpos( buf, 'b' );
        if( n == -1 )
            printf("The character was not found. \n");
    }
}

```

```

else
    printf("The last matched character position got by strrpos( ): %d. \n", n);
}
else
    printf("The character wasn't found. \n");

return(0);
}

```

### A.5.16 函数名: strspn, strcspn, strpbrk, strrpbrk

原型:

```

extern int strspn(char *s1, char *set);
extern int strcspn(char *s1, char *set);
extern char *strpbrk(char *s1, char *set);
extern char *strrpbrk(char *s1, char *set);

```

功能:

对于 strspn( ) 在 s<sub>1</sub> 串找出不包含在集合 set 中的字符, 返回 s<sub>1</sub> 中第一次不包含在 set 中的字符位置, 如 s<sub>1</sub> 中未发现含在 set 中的字符, 则返回 s<sub>1</sub> 的全长(不包括结束符)。

strcspn 与 strspn 相反, 但它搜索的是 s<sub>1</sub> 串中第一个包含在集合 set 里的字符。

strpbrk 与 strcspn 很相似, 但返回的是 s<sub>1</sub> 中第一个包含在集合 set 中字符的指针, 而不是个数, 如果未找到, 则返回 NULL。

strrpbrk 与 strpbrk 同是在 s<sub>1</sub> 中找寻包含在集合 set 中的字符, 但找的不是第一个, 而是最后一个含在集合 set 中的字符, 并返回这个字符的指针。如果 s<sub>1</sub> 中没有含在集合 set 中的字符, 则返回 NULL 指针。

[例 A.60]

```

test _strspn _strcspn _strpbrk _strrpbrk( )
#include <string.h>
#include <stdio.h> /* for printf */
int main(void)
{
    char *digit_str = "12348967";
    char octd[ ] = "01234567";
    char vowels = "AEIOUaeiou";
    char text[ ] = "Once upon a time...";
    int n;
    char *p;

    /* strspn() */
    n = strspn(digit_str, octd);
    if (digit_str[n] != '\0')
        printf("%c is not a octal digit at index %d of digit_str. \n", digit_str[n], n);

    /* strcspn( ) */
}

```

超星浏览器提醒您:  
使用本复制品  
请尊重相关知识产权!

```

strcpy(digit_str, "12345.6789");
n = strcspn(digit_str, '.');
if (digit_str[n] != '\0')
    printf("%c was found at index %d in %s. \n", (char) digit_str[n], n, digit_str);

/* strchr() */
p = strchr( text, vowels );
if(p == NULL)
    printf("No vowels were found in %s. \n", text );
else
    printf("The found vowel is %c at index %d of %s. \n", *p, text - p, text );

/* strrchr() */
p = strrchr( text, vowels );
if(p == NULL)
    printf("No vowels were found in %s. \n", text );
else
    printf("The last vowel is %c at index %d of %s. \n", *p, text - p, text );

return(0);

```



## A.6 字符串转换函数

本节函数的原型说明包含在头文件 `stdlib.h` 中。

### A.6.1 函数名: `atof`

原型:

```
extern double atof(char *s1);
```

功能:

`atof` 将 `s1` 串转换为浮点数, 并返回它。输入串必须包含与浮点数规定相符的字符数。C51 中对类型 `float` 和 `doudle` 等同对待。

[例 A.61]

```

test_atof( )
#include <stdlib.h>
#include <stdio.h> /* for printf */
int main(void)
{ float f;
  char *str = "12234.567";

  f = atof(str);
  printf("string = %s, float = %f. \n", str, f);
}

```

```
return(0);
```

### A.6.2 函数名: atol

原型:

```
extern long atol(char *s1);
```

功能:

atol 将 s<sub>1</sub> 串转换为长整型数, 并返回它。输入串必须包含与长整型规定相符的字符数。

[例 A.62]

```
test_atol( )
#include <stdlib.h>
#include <stdio.h> /* for printf */
int main(void)
{ long l;
  char *lstr = "98761234";

  l = atol(lstr);
  printf("string = %s, integer = %ld.\n", lstr, l);

  return(0);
}
```

### A.6.3 函数名: atoi

原型:

```
extern int atoi(char *s1);
```

功能:

atoi 将串 s<sub>1</sub> 转换为整型数, 并返回它。输入串必须包含与整数规定相符的字符数。

[例 A.63]

```
test_atoi( )
#include <stdlib.h>
#include <stdio.h> /* for printf */
int main(void)
{ int n;
  char *str = "12345";

  n = atoi(str);
  printf("string = %s, int = %d.\n", str, n);

  return(0);
}
```

超星阅读器提醒您：  
使用本复制品  
请尊重相关知识版权！



## A.7 变参数函数

本节函数的原型说明在头文件 `stdarg.h` 中。

C51 允许再入函数使用变参数(变参数在参数表中记做“...”)。在编译时变参数的个数和数据类型是未知的,为此,在头文件 `stdarg.h` 中定义了下列宏。

### A.7.1 宏名: `va_list`

功能:

它是一个自定义的数组类型,用以存放变参数信息表。

### A.7.2 宏名: `va_start(va_list ap, last_argument)`

功能:

初始化变参数信息表。本参数宏有两个参数:变参数信息数组指针和函数参数表的最后一个固定参数名。

### A.7.3 宏名: `type va_arg(va_list ap, type)`

功能:

本参数宏有返回类型。它是函数的缺省返回类型。实际上,它是 `int` 的扩展类型,包括 `int`, `unsigned int` 和 `double`。本参数宏每调用一次返回一次变参数信息表中的下一个参数。调用时,除变参数信息表指针外,还有下一个参数的类型。

### A.7.4 宏名: `va_end(va_list ap)`

功能:

变参数信息表中的参数均已用完时,调本参数宏修改 `ap` 使再次调用 `va_start()` 前不被使用。

[例 A.64]

```
test_va_start_srg_end()
#include <stdarg.h>
#include <stdio.h>    /* for printf */

/* 计算连续输入 int 数的总和,以输入 0 作为结束。定义一具有变参数的函数 sum( ) */
void    sum(char * msg, ...)
{
    int    total = 0;
    va_list ap;
    int    arg;

    va_start(ap, msg);
    while((arg = va_arg(ap, int)) != 0)
        total += arg;
```

```
printf(msg, total );
va _end( ap );
```

```
int      main(void)
```

```
sum(" The total of a series of integers ending with a "0" is %d.", 3,4,5,6,7,0);
return(0);
```

程序输出:

The total of a series of integers ending with a "0" is 25.

[例 A.65]

```
#include <stdarg.h>
```

```
#include <stdio.h>    /* for printf */
```

```
void      pf(int a, ...)
```

```
va _list  argptr;
```

```
va _start(argptr, a );
```

```
while( a -- )
```

```
puts( va _arg(argptr, char * ) );
```

```
va _end( argptr );
```

```
int      main(void)
```

```
pf( 3, "line1 \n", "line2 \n", "line3 \n" );
```

```
return(0);
```

程序输出:

line1

line2

line3

[例 A.66]

```
#include <stdarg.h>
```

```
#include <stdio.h>    /* for printf */
```

```
int      varfunc(char * buf, int id, ...)
```

```
va _list;
```

```
va _start(tag , id );
```

```
if( id == 0 )
{
    int  arg1;
    char *arg2;
    long arg3;

    arg1 = va_arg(tag, int);
    arg2 = va_arg(tag, char *);
    arg3 = va_arg(tag, long);
}
else
{
    char *arg1;
    char *arg2;
    long  arg3;

    arg1 = va_arg(tag, char *);
    arg2 = va_arg(tag, char *);
    arg3 = va_arg(tag, long);
}

va_end( argptr );
}
```

```
int      main(void)
{
    char  buf[20];

    varfunc(buf, 0, 33, test, 123L);
    varfunc(buf, 1, debug, test, 123456L);

    return(0);
}
```



## A.8 全程跳转函数

本节函数的原型说明在头文件 `setjmp.h` 中。本节的 `setjmp` 和 `longjmp` 函数联合使用实现非本地跳转(`goto`)。它允许从深层函数调用中直接返回外层函数。

### A.8.1 函数名: `setjmp`

原型:

```
extern int setjmp(jmp_buf jpbuff);
```

功能:

将当前状态信息存于 `jpbuff` 中, 供函数 `longjmp` 使用。直接调本函数, 返回 0, 由 `longjmp` 调用, 返回非 0 值。只能在 `if` 或语句中调用一次。

### A.8.2 函数名: `longjmp`

原型:

```
extern void longjmp(jmp_buf jpbuff, int val);
```

功能:

将调用 setjmp( ) 时存于 jpbufl 中的状态恢复, 并以参数 val 值取代 setjmp( ) 的返回值返回给原调用 setjmp 的函数。这时原调用函数的自动变量和未说明为 volatile 的变量值均已改变, 应予注意。

setjmp 和 longjmp 常常协同用于处理低层函数调用遇到的错误和意外。

[例 A. 67]

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void inner( )
{
    longjmp(jb, 5);
}

void main( )
{
    int i;
    if(i = setjmp(jb))
    {
        printf("setjmp returned %d \n", i);
        exit(0);
    }
    printf("longjmp returned 0 --- good! \n");
    printf("calling inner \n");
    inner( );
}
```

[例 A. 68]

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

void subroutine(jmp_buf jpbufl )
{
    longjmp(jpbufl, 2);
}

void main( )
{
    int val;
```



```

jmp_buf jbuf;

if( val = setjmp(jbuf) )
{
    printf("setjmp returned from subroutine with %d \n", val);
    exit(val);
}

printf("setjmp returned 0 -- good! \n");
printf("about to call subroutine... \n");
subroutine(jbuf);

```

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

## A.9 内部函数

本节函数的原型说明在头文件 `intrins.h` 中。

### A.9.1 函数名：`_crol_`，`_irol_`，`_lrol_`

原型：

```

unsigned char _crol_(unsigned char val, unsigned char n);
unsigned int _irol_(unsigned int val, unsigned char n);
unsigned long _lrol_(unsigned long val, unsigned char n);

```

功能：

这几个函数都将 `val` 左移 `n` 位。不同函数参数类型不同。

[例 A.69]

```

#include <intrins.h>

void main( )
{
    unsigned int y;
    y = 0x00ff;
    y = _irol_(y, 4);      /* y 值为 0x0ff0. */
}

```

### A.9.2 函数名：`_cror_`，`_iror_`，`_lror_`

原型：

```

unsigned char _cror_(unsigned char val, unsigned char n);
unsigned int _iror_(unsigned int val, unsigned char n);
unsigned long _lror_(unsigned long val, unsigned char n);

```

功能：

这几个函数都将 `val` 右移 `n` 位。

[例 A.70]

```

#include <intrins.h>

```

```

void main( )
{
    unsigned int y;

    y = 0xff00;
    y = _iror_(y, 4);    /* y 值为 0x0ff0 */
}

```

### A.9.3 函数名: \_nop\_

原型:

```
void _nop_(void);
```

功能:

\_nop\_ 产生一个 NOP 指令。

[例 A.71] 从 P0.7 输出三个机器周期宽的正脉冲。

```
P0  _&= 0x80
```

```
P0  _1= 0x80
```

```
_nop_
```

```
_nop_
```

```
P0  _&= 0x80
```

### A.9.4 函数名: \_testbit\_

原型:

```
bit _testbit_(bit x);
```

功能:

\_testbit\_ 产生一条 JBC 指令。该函数测试位变量, 当置位时返回 1, 否则返回 0。如果该位置为 1, 还在测试后将该位复位为 0。这和 8051 的 JBC 指令作用一样。\_testbit\_ 只能作用于直接寻址的位变量; 不能用于表达式中。

[例 A.72]

```

1    #include<intrins.h>
2
3    char val;
4    bit flag;
5
6    main( ) {
7    1  if(!_testbit_(flag)) val--;
8    1 }

```

ASSEMBLY LISTING OF GENERATED OBJECT CODE

```
; FUNCTION main(BEGIN)
```

```
; SOURCELINE# 6
```

```
; SOURCELINE# 7
```

```
0000 100002 R JBC flag, ? C0002
```



```

0003    1500    R    DEC    val
                        ; SOURCE LINE # 8
0005          ? C0002:
0005 22          RET
                        ; FUNCTION    main(END)

```

## A.10 抽象数组

本节函数的原型说明放在头文件 `absacc.h` 中。

### A.10.1 函数名: CBYTE, DBYTE, PBYTE, XBYTE

原型:

```

#define CBYTE ((unsigned char *)0x50000L)
#define DBYTE ((unsigned char *)0x40000L)
#define PBYTE ((unsigned char *)0x30000L)
#define XBYTE ((unsigned char *)0x20000L)

```

功能:

上述宏定义用做对各种存储空间按 `char` 数据类型进行绝对地址访问。`CBYTE` 访问 `CODE` 空间, `DBYTE` 访问 `DATA` 空间, `PBYTE` 访问 `XDATA` 空间第一页, (通过 `MOVX (R0)` 访问), `XBYTE` 访问比例 `XDATA` 空间(通过 `MOVX @DPTR` 访问)。

[例 A.73] 下列语句访问外部数据空间 `0x1000` 地址处的字节变量。

```

XBYTE[0x1000] = 20;
XVAL = XBYTE[0x1000];

```

### A.10.2 函数名: CWORD, DWORD, XWORD, PWORD

原型:

```

#define CWORD ((unsigned int *)0x50000L)
#define DWORD ((unsigned int *)0x40000L)
#define PWORD ((unsigned int *)0x30000L)
#define XWORD ((unsigned int *)0x20000L)

```

功能:

上述宏定义用做对各种存储空间按 `char` 数据类型进行绝对地址访问。其他与上节同。

[例 A.74] 下列语言访问外部数据空间 `0x1000` 地址处的字变量。

```

XWORD[0x1000] = 20;
XVAL = XWORD[0x1000];

```





# 附录 B C51 编译器使用错误提示



## B.1 前言

C51 能识别的错误种类:

- 致命错误
- 语法及语义错误
- 警告

1. 致命错误发生, 立即终止编译。错误原因一般是: 伪指令控制行有错; 访问不存在的源文件或头文件等。

2. 语法和语义错都发生在源文件中。有这类错误, 给出提示但不再产生目标文件, 错误超过一定数量才终止编译。

3. 警告出现并不影响目标文件的产生, 但执行时可能会发生问题。程序员应自己斟酌处理。

## B.2 致命错误

致命错误有如下两种格式:

C\_51 FATAL\_ERROR  
ACTION: <当前行为>

C\_51 FATAL\_ERROR  
ACTION: <当前行为>

分析源程序时发现外部引用名太多。

- GENERATING INTERMEDIATE CODE

源代码被翻译成内部伪代码,错误可能来源于函数太大超过内部极限。

- WRITING TO FILE

在向文件(work, list, prelist 或 object file)写时发生错误。

ERROR 有下列信息:

- MEMORY SPACE EXHAUSTED

所有可用系统空间耗尽。至少需要 512K 字节空间。如果有足够空间,用户必须检查常驻内存的驱动程序是否太多。

- FILE DOES NOT EXIST

FILE 行定义的文件名未发现。

- CAN'T CREATE FILE

FILE 行定义的文件不能被创建。

- SOURCE MUST COME FROM A DISK FILE

源文件和头文件必须存在于硬盘或软盘上。控制台:CON:, :CI:或类似设备不允许作为输入文件。

- MORE THAN 256 SEGMENTS / PUBLICS / EXTERNALS

受 OMF-51 的历史限制,一个源程序不能超过 256 个各种函数的类型段、256 个全局变量、256 个公共定义或外部引用名。不使用位变量可减少使用的段数。使用 static 存储类说明符可减少全局变量的使用数。合理调整定义性说明的位置可减少外部引用名的使用数。

#### FILE WRITE ERROR

当向 list, preprint, work 或 object 文件中写内容时,由于空间不够而发生错误。

- NON\_NULL ARGUMENT EXPECTED

所选的控制参数需要一个括号内的变量,如一个文件名或一个数。

- UNKNOWN CONTROL

所选的控制参数是未知的。

- "(" AFTER CONTROL EXPECTED

有些控制参数需要用括号括起的变量,左括号丢失。

- ")" AFTER PARAMETER EXPECTED

变量的右括号丢失。

- RESPECIFIED OR CONFLICTING CONTROL

所选的控制参数与前面发生冲突或重复,例如 CODE 和 NOCODE。

- BAD DECIMAL NUMBER

控制参数的数字含有非法数,需要使用十进制数。

- OUT OF RANGE DECIMAL NUMBER

控制参数的数字越界,例如 OPTIMIZE 的参量为 0~5。

- IDENTIFIER EXPECTED

控制参数 DEFINE 需要一个标识符做参量,与 C 语言的规则相同。

- PARSE STACK OVERFLOW



分析栈溢出。可能是源程序包含特别复杂的表达式,或功能块嵌套数超过 15。

- MORE THAN 100 ERRORS IN SOURCE\_FILE

编译期间检测到 100 个错误,引起编译终止。

- PREPROCESSOR:MACRO TOO NESTED

宏扩展期间,预处理器的栈耗用太大。表明宏嵌套太多或有递归宏定义。

- PREPROCESSOR:LINE TOO LONG(510)

宏扩展后行超过 510 个字符。

- CAN'T HAVE GENERAL CONTROL IN INVOCATION LINE

一般控制(如 EJECT)不能是命令行的一部分,应将它们放入源文件“pragma”预处理行中。

### B.3 语法及语义错误

1. 这类错误在列表文件中产生如下格式的信息:

```
* * * ERROR <number> IN LINE <line> OF <file>:error message
```

```
* * * WARNING <number> IN LINE <line> OF <file>:warning message
```

其中:

<number>表示错误号。

<line>表示源文件或头文件中与错误或警告相关的行。

<file>指明了错误所在的源文件或头文件。

error message 或 warning message 的信息结构依赖于所遇错误的类型。对于语法错误会指明期望的语法,对于语义错误会显示出与错误有关的对象符号名。

错误和警告信息的例子:

```
* * * ERROR 202 IN LINE 19 OF TEST. C: "i":undefined identifier
```

```
* * * ERROR 141 IN LINE 10 OF TEST. C: NEAR "|" EXPECTED';'
```

```
* * * WARNING 206 IN LINE 102 OF CLOCK. C: "printf" missing function -
prototype
```

2. 下面给出错误信息及可能发生的原因。

- ERROR 100:unprintable character 0x?? skipped

源文件中发现非法字符(注意注解内的字符不作值检查)

- ERROR 101:unclosed string

串未用引号结尾。

- ERROR 102:string too long

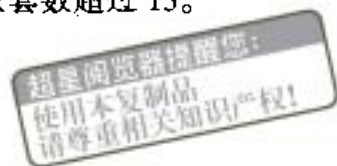
串不得超过 511 个字符,为了定义更长的串,用户必须使用续行符‘\’逻辑地继续该串,在词汇分析时遇到以该符号结尾的行会与下行连接起来。

- ERROR 103:invalid character constant

字符常数格式无效,记号‘\C’有效仅当‘C’为可打印的 ASCII 字符。

- ERROR 104:macro redeclaration

试图再声明一个已定义的宏,已存在的宏可用 #undef 指令删除。预定义的宏不能删除。



编译器识别下列预定义的宏：

\_DATE\_  
\_STDC\_  
\_LINE\_  
\_FILE\_

- ERROR 105 :identifier expected

预处理器语法期望一个标志符,如 `ifdef <name>`。

- ERROR 106: unclosed comment

当注解无结束定界符(`*/`)时产生此错误。

- ERROR 107:unbalanced #if-endif controls

`endif` 的数量与 `if` 或 `ifdef` 的数量不匹配。

- ERROR 108:include file nesting exceeds 9

嵌套的头文件超过最大值 9。

- ERROR 109:bad or missing include file name

`include` 指令后的文件名无效或丢失。

- ERROR 110:expected string

预处理器指令期望一个串变量,如 `#error "string"`。

- ERROR 111: <user error text>

由 `#error` 伪指令引入的错误信息以错误号形式显示。

- ERROR 112:missing directive

预处理行 `#` 号后缺少伪指令。

- ERROR 113:unknown directive

预处理行 `#` 号后的量不是伪指令。

- ERROR 114:misplaced 'elif'

- ERROR 115:misplaced 'else'

- ERROR 116:misplaced 'endif'

指令 `elif/else/endif` 只有在 `if,ifdef` 或 `ifndef` 指令内才是合法的。

- ERROR 117:bad integer expression

`if/elif` 指令的数值表达式有语法错误。

- ERROR 118:missing '(' after macro identifier

宏调用中实参表或包括实参表的左括号丢失。

- ERROR 119:reuse of macro formal parameter

宏定义的形参名重复使用。

- ERROR 120: 'C' unexpected in formal list

形参表中不允许有字符 'C',应用逗号代替。

- ERROR 121:missing ')' after actual parameters

宏调用实参表的右括号丢失。

- ERROR 122:illegal macro invocation

宏调用的实参表与宏定义中的形参表不同。



- ERROR 123: missing macro name after 'define'

# define 伪指令后缺欲定义的宏名。

- ERROR 124: expected macro formal parameter

宏定义要求形参名。

- ERROR 125: declarater too complex(20)

说明符过于复杂。

- ERROR 126: type - stack underflow

对象的声明最多只能包含 20 个类型修饰符( [ ], \*, ( ) )。错误 126 经常在错误 125 之前, 两者一起发生。

- ERROR 127: invalid storage class

对象被用无效的存贮类所说明。当在函数外用 auto/register 存贮类时会发生这种情况。

- ERROR 128: memory space: illegal memory space, 'memory space' used

函数参数的存贮类由存贮模式(SMALL, COMPACT, LARGE)决定, 用户不能改变, 使用不同于存贮模式的自动变量应改为静态的存贮类。

- ERROR 129: missing ';' before 'token'

该错误表明分号丢失, 通常该错误后会有引发一连串错误, 引发的这些错误信息无关紧要。因为缺少分号后, 编译器不能作正确的语法分析。

- ERROR 130: value out of range

'using' 或 'interrupt' 指令后数值越限。'using' 用的寄存器组号为 0~3, 'interrupt' 需要 0~15 的中断号。

- ERROR 131: duplicate function - parameter

函数中形参名重复。形参名应彼此不同。

- ERROR 132: not in formal parameter list

函数内参数声明使用的名字未出现在参数表中。

- ERROR 133: char function (v0, v1, v2)

```
char * v0, * v1, * v5; / * 'v5' 在形参表中未出现 * /
|
/ * ... * /
|
```

- ERROR 134: xdata / idata / pdata / data on function not permitted

函数总是驻留于 0x5xxx 的 CODE 存贮区, 不能位于 xdata / idata / pdata / data 空间。

- ERROR 135: bad storage class for bit

位变量的定义可以接受 static 或 extern 的存贮类。用 REGISTER 和 ALIEN 存贮类都是非法的。

- ERROR 136: 'void' on variable

'void' 类型只允许作为函数的返回类型或与指针类型使用(void \*)。

- ERROR 137: illegal parameter type: 'function'

函数参数的类型不能是函数, 然而函数指针可作为参数。

- ERROR 138: interrupt ( ) may not receive or return value (s)

中断函数既不能有参数也不能有返回值。

- ERROR 139: illegal use of 'alien'

关键字 'alien' 将函数定义为 PL/M51 规定的过程与函数的结构。这意味着 C 函数中有变参数的缩记符号 (即 `func(...);`) 是不能使用 'alien' 的。

- ERROR 140: bit in illegal memory - space

位变量的定义可包含修饰符 DATA, 如果无修饰符则假定为 DATA。因为位变量始终位于 0x4xxx 的内部数据存储器中, 当试图采用其他存储空间时, 会产生这个错误。

- ERROR 141: NEAR <token>: expected <token>, ...

编译器所见的单词是错误的。期望正确的单词。

- ERROR 142: invalid base address

SFR 说明中的基址有错。有效基址为 0x80~0xFF。如果声明采用 'base`pos' 形式, 则基址是 8 的整数倍。

- ERROR 143: invalid absolute bit address

sbit 说明中位地址必须在 0x80~0xFF 间。

- ERROR 144: base ^ pos: invalid bit position

sbit 说明中位 pos 必须在 0~7 之间。

- ERROR 145: undeclared sfr

sfr 未说明。

- ERROR 146: invalid sfr

绝对位址的说明 (base`pos) 包含无效的基地址。这个基地址必须与 SFR 名相对应。

- ERROR 147: object too large

对象不能超过 65 536 (64K) 字节。

- ERROR 148: field not permitted in union

联合不能包含位成员, 这个限制是由 8051 结构产生的。

- ERROR 149: function member in struct / union

结构或联合不能包含函数类型的成员。但指向函数的指针是允许的。

- ERROR 150: bit member in struct / union

结构或联合不能包含位类型的成员, 这个限制是由 8051 结构决定的。

- ERROR 151: self relative struct / union

结构或联合不能包含自身。

- ERROR 152: bit - field type too small for number of bits

位域声明中指定的位数超过所给原型中位的数量。

- ERROR 153: named bit - field cannot have 0 width

命名的域宽度为 0, 只有未命名的位域允许有 0 宽度。

- ERROR 154: ptr to field

无指向位域指针的类型。

- ERROR 155: char / int required for fields

位域基类型要求 char 或 int 类型, unsigned char 或 unsigned int 也有效。

- ERROR 156: alien permitted on function only



alien 只能用于函数。

- ERROR 157: var \_ parms on alien function

有变参数的函数不能用 'alien'。因 PL/ M-51 函数只能用固定数量的参数。

- ERROR 158: function contains unnamed parameter

函数定义的参数表中包含无名参数。无名参数只允许用于函数的原型中。

- ERROR 159: type follows void

函数原型声明中可含一个空的参数表 'f (void)'。void 后不能再含有其他类型定义。

- ERROR 160: void invalid

void 类型只能与指针合用或表明函数无返回值。

- ERROR 161: formal parameter ignored

函数内的外部函数引用声明使用了无类型的参数表,例如 'extern yy(a, b, c);'。要求形参表。

- ERROR 162: duplicate function - parameter

函数内参数名重复。

- ERROR 163: unknown array size

不管是一维数组还是多维数组或外部数组,都需要指定数组的大小,这个大小由编译器在初始化时刻计算。这个错误是试图为一未定维的数组使用 'sizeof' 运算符的结果,或者是一个多维数组附加元素未定义的结果。

- ERROR 164: ptr to null

这个错误通常是前一个错误造成的结果。

- ERROR 165: ptr to bit

指向位的指针是不合法的类型。

- ERROR 166: array of functions

数组不能包含函数,但可包含指向函数的指针。

- ERROR 167: array of fields

位域不能安排为数组。

- ERROR 168: array of bit

数组没有位类型。

- ERROR 169: function returns function

函数不能返回函数,但可返回一个指向函数的指针。

- ERROR 170: function returns array

函数不能返回数组,但可返回指向数组的指针。

- ERROR 171: missing enclosing loop

'break' 或 'continue' 语句只能出现在 for, while, do while 或 switch 语句中间。

- ERROR 172: missing enclosing switch

case 语句只能出现在 switch 语句中。

- ERROR 173: missing return - expression

返回值类型不是 integer 的函数必须包含一条带表达式的 return 语句。由于要与老版本兼容,编译器对返回整型值的函数不作检查。



- ERROR 174: return - expression on void - function

void 函数不能返回值, 因此不能包含带表达式的 return 语句。

- ERROR 175: duplicate case value:

每个 case 语句必须包含一个常量表达式作其变量, 这个值不能在 SWITCH 语句的给定级中出现多次。

- ERROR 176: more than one 'default'

SWITCH 语句中不能包含多于一个的 default 语句。

- ERROR 177: different struct / union

赋值或参数传递中使用了结构/联合的不同类型。

- ERROR 178: struct/union comparison illegal

根据 ANSI C, 两个结构或联合的比较是不允许的。

- ERROR 179: can't / cast from / to void - type

从 'void' 类型转进或转出都是非法的。

- ERROR 180: can't cast to 'function'

转换为 'function' 类型是非法的, 使用函数指针指向不同的函数。

- ERROR 181: incompatible operand

在所给的运算符中至少有一个操作符类型是无效的。

- ERROR/WARNING 182: pointer to different objects

指针使用的不一致性。

- ERROR 183: unmodifiable value

欲修改的对象位于 CODE 存储区, 因而不可修改。

- ERROR 184: sizeof: illegal operand

sizeof 运算符不能决定函数或位域大小。

- ERROR 185: different memory space

对象说明的存储器空间与前面的不一致。

- ERROR 186: invalid dereference

这条错误信息可能是由编译器内部问题产生的。

- ERROR 187: not an lvalue

所需参量必须是可变对象的地址。

- ERROR 188: unknown object size

无法计算对象大小, 因为缺少数组维数或因为是通过 void 指针的间接访问。

- ERROR 189: '&' on bit / sfr illegal

地址操作符 '&' 不允许用于位对象或 SFR。

- ERROR 190: '&': not an lvalue

地址不是可变的对象不能作为左值。

- ERROR 191: '&' on constant

试图为所列类型常数建立指针。

- ERROR/ WARNING 192: '&' on array/ function

地址操作符 '&' 不允许用于数组和函数。函数和数组本身都代表了地址。



- ERROR 193: illegal op - type(s)
- ERROR 193: illegal add/sub on ptr
- ERROR 193: illegal operation on bit (s)
- ERROR 193 : bad operand type

当一个表达式使用给定运算符的非法操作类型时会出现该错误。无效的表达式, 例如 bit + bit, ptr + ptr 或 ptr \* <any>。错误信息包括引起错误的运算符。

下列运算可使用位操作符:

赋值( = )  
 OR/复合 OR( | , | = )  
 AND/复合 AND( & , & = )  
 XOR/复合 XOR( ^ , ^ = )  
 位或常数的按位比较( == ! = )  
 取反( ~ )

位类型运算符可和其他数据类型一起在表达式中使用。这种情况下自动进行类型转换。

- ERROR 194: ' \* ' indirection to object of unknown size

间接操作符 ' \* ' 不能用于 void 指针 (void \* ), 因为指针所指的对象大小是未知的。

- ERROR 195: ' \* ' illegal indirection

间接操作符 ' \* ' 不能用于非指针变量。

- ERROR /WARNING 196 : mspace probably invalid

产生此警告是由于将某些常数值赋给指针并且常数没有形成一个有效的指针值, 有效的指针常数类型为 long/unsigned long。编译器对指针对象采用 24bits(3 字节), 低 16 位表示偏移, 高 8 位表示存储类的选择, 在低字节中, 值从 1~5 表明了 XDATA, PDATA, IDATA, DATA 和 CODE 存储类。

- ERROR 197: illegal pointer assignment

试图将一个非法对象赋给指针, 只有另一个指针或指针常量可以赋给指针。

- ERROR /WARNING 198 : size of returns zero

求某些对象长度得到 0。如果对象是外部的或一个数组中不是所有维的大小都是已知的时得到 0, 这时候该值可能是错的。

- ERROR 199: left side of ' - > ' requires struct / union pointer

' - > ' 操作符的左边变量必须是结构或联合的指针。

- ERROR 200: left side of ' . ' requires struct / union

' . ' 操作符的左边变量必须具有结构/联合类型。

- ERROR 201: undefined struct/ union tag

所给的结构/联合标记名是未知的。

- ERROR 202: undefined identifier

所给的标识符未定义。

- ERROR 203: bad storage class(nameref)

该错误表明编译器内部问题。

- ERROR 204: undefined member



所给的结构/联合成员名未定义。

- ERROR 205: can't call an interrupt function

中断函数不能象普通函数一样调用,因为这类函数的头段和尾段是为中断特殊编码的。

- ERROR / WARNING 206: missing function - prototype

调用的函数缺少原型声明。这条信息可以理解为一个警告,调用未知函数总要冒形参与实参不相符的风险,编译器对缺少或多出的参数及其类型未作检查,因而函数不能正确调用。用户应该在源程序的开头说明欲调用的函数的原型。函数的定义性说明会自动产生一个原型,所以,也可以把函数的定义性说明放在源程序的开头。

- ERROR 207: declared with 'void' parameter list

用 void 参数说明的函数不接受调用者传来的参数。

- ERROR 208: too many actual parameters

函数调用包含了多余的实参。

- ERROR 209: too few actual parameters

函数调用时传递的实参过少。

- ERROR 210: too many nested calls

超过了 10 个函数嵌套调用的极限。

- ERROR 211: call not to a function

函数调用时没有函数的地址或未对指向函数的指针赋值。

- ERROR 212: indirect call with parameters

由于参数传递方法的限制,通过指针的间接函数调用不能作为实参。这种参数传递方法要求被调用的函数名已知,因为参数要被写入调用函数的数据段。然而,间接调用时被调用函数的名字是未知的。

- ERROR 213: left side of assign \_ op not an lvalue

在赋值操作符的左边要求可变的对象。

- ERROR 214: can't cast non \_ pointer to pointer

非指针不能转换为指针。

- ERROR 215: can't cast pointer to non \_ int / pointer

指针可以转换为另一个指针或整数,但不能转换为其他类型。

- ERROR 216: subscript on non \_ array or too many dimensions

对非数组使用了下标或数组维数过多。

- ERROR 217: non \_ integral index

数组的下标表达式必须是整型类型:即 char, unsigned char, int 或 unsigned int。其他类型均是非法的。

- ERROR 218: void \_ type in controlling expression

while, for 或 do while 语句中的条件表达式不能是 void 类型。

- ERROR 219: long constant truncated to int

企图把长整型常量截断为整型数是错的。

- ERROR 220: illegal constant expression

非法常数表达式。

• ERROR 221: non \_ constant case/ dim expression  
case 值或下标值([ ])要求用常数表达式。

• ERROR 222: div by zero

• ERROR 223: mod by zero

编译器检测到 0 除或 0 模的错误。

• ERROR 224: illegal operation on float/double

AND 和 NOT 一类的运算符不允许作用于 float/double 变量。

• ERROR 225: expression too complex, simplify

表达式太复杂, 必须简化。

• ERROR 226: duplicate struct/ union /enum tag

duplicate struct/ union /enum 重复标记。

• ERROR 227: not a union tag

所给的标记名虽已定义, 但不是联合的标记。

• ERROR 228: not a struct tag

所给的标记名虽已定义, 但不是结构的标记。

• ERROR 229: not an enum tag

所给的标记名虽已定义, 但不是枚举类型的标记。

• ERROR 230: unknown struct/union /enum tag

所给的 struct/union /enum 标记名未定义。

• ERROR 231: redefinition

所给的名字已经定义, 不能再被定义。

• ERROR 232: duplicate label

所给的标号已经定义。

• ERROR 233: undefined label

当对函数进行分析后, 编译器检查函数有未定义的标号, 发出错误信息。

• ERROR 234: '}' scope stack overflow(31)

超过了最大为 31 个的功能块嵌套极限, 多余的块被忽略。

• ERROR 235: parameter <number> : different types

函数实参类型与函数原型中的不同。

• ERROR 236: different length of parameter lists

所给的函数实参数量与函数原型中的不同。

• ERROR 237: function already has a body

试图定义已定义过的函数。

• ERROR 238: duplicate member

• ERROR 239: duplicate parameter

重复定义结构成员或函数参数。

• ERROR 240: more than 128 local bit 's

位变量定义总数不得超过 128。

• ERROR 241: auto segment too large



局部对象要求的空间超过了该模式的最大值。最大栈长定义如下:

SMALL:128 字节; COMPACT: 256 字节; LARGE: 64K。

• ERROR 242: too many initializers

初始化对象数量超限。

• ERROR 243: string out of bounds

串中字符数超过了字符数组要求初始化的字符数。

• ERROR 244: can't initialize, bad type or class

试图初始化位或 SFR。

• ERROR 245: unknown pragma, line ignored

未知 pragma 语句, 因此该行被忽略。

• ERROR 246: floating point error

本错误发生在浮点变量超过 32 位有效字长时, 32 位 IEEE 格式浮点值的范围是 (1.175494E-38~3.402823E+38)。

• ERROR 247: non \_address + / - constant initializer

有效的初始化表达式必须是非地址量 + / - 常量。

• ERROR 248: aggregate initialization needs curly braces

所有的组合变量(数组、结构或联合)初始化时要用花括号括起来。

• ERROR 249: segment <name>: segment too large

编译器检测到过大的数据段, 最大数据段长决定于存贮器空间。

• ERROR 250: '\ esc'; value exceeds 255

串常数中 \ esc 转义序列的值超过有效值域(最大为 255)。

• ERROR 251: illegal octal digit

不是有效的八进制数字。

• ERROR 252: misplaced primary control, line ignored

一次性使用的编译控制伪指令必须在 C 模块开头指定, 在 # INCLUDE 语句和变量说明之前。

• ERROR 253: internal ERROR (ASMGGEN \ CLASS)

这种错误在下列情况下发生:

• 内部函数(如 testbit)被不正确激活。它发生在函数原型和实参表不存在匹配问题的时候。基于这个原因, 头文件的使用要适当(intrins.h, string.h)。

• C51 识别出存在内部一致性错误, 请向你的销售代理商查询。

• ERROR 255: switch expression has illegal type

switch 语句中的 case 语句必须具有类型(u)char, (u)int 或(u)short, 其他类型不允许(如 bit)。

• ERROR 256: conflicting memory model

含“alien”属性的函数只能使用“small”模式。函数的参数必须位于内部数据存贮空间中。这也适用于外部“alien”声明和“alien”函数, 如:

```
alien plmstyle(char C)large {...}      / * ERROR, 256 * /
```

• ERROR 257: ailen function can not be reentrant



含有“alien”属性的函数不能同时具有“reentrant”属性。函数的参数不能通过重入栈传递,这也适用于外部“alien”声明和“alien”函数。

- ERROR 258: mspace illegal on struct/union member

不能为结构成员指定存储空间,但指向对象的指针可以,如:

```
struct vp {char code c; int xdata i;}; /* ERROR 258 */
struct vp {char c; int xdata * i;}; /* Correct */
```

- ERROR / WARNING 259: pointer: different mspace

当为指针赋值或做指针比较时,指针未指向同一存储空间的对象时,会产生错误或警告,如:

```
char xdata * px; /* px to char in xdata memory */
char code * pc; /* pc to char in code memory */
void main()
{
    char C;
    if( px == pc) + + c; /* warning 259 */
    px = pc; /* warning 259 */
}
```

- ERROR / WARNING 260: pointer truncation

指针转换时部分偏移被截断,此时指针常量(如 char xdata)转为一个具有较小偏移区的指针(如 char idata)。

- ERROR 261: bit in reentrant function

重入函数不能包含位变量,因为位变量不能存于重入栈,而只能位于 MCS-51 CPU 的可位寻址存储区中,如:

```
void test() reentrant
{
    bit b0; /* illegal */
    static bit b1; /* legal */
    ...
}
```

- ERROR 262: 'using /disable': function returns bit

使用属性“using”选择寄存器组的函数或使用关中断( #pragma disable )功能的函数不能返回“bit”类型。例如:

```
bit test ( ) using 3 /* ERROR 262 */
{
    bit b0
    return (b0);
}
```

- ERROR 263: save - stack overflow/ underflow

“ #pragma save”最大嵌套深度为 8 级。SAVE 和 RESTORE 指令以 LIFO 原则工作。

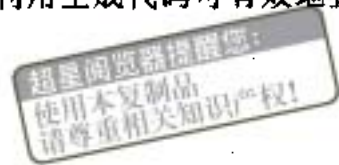
- ERROR 264: intrinsic <intrinsic\_name> ; declaration/activation error

内部函数定义不正确(参数数量或省略号)。

办法:不要为内部函数或库函数定义原型,应使用头文件。

·WARNING 265: <name> : recursive call to non\_reentrant function

发现非重入函数被递归调用。直接递归有意识地利用生成代码可有效地查出,间接递归由 L51 发现。





## 附录 C L51 连接/定位器使用错误提示



### C.1 前 言

L51 能识别的使用错误种类：

- 警告
- 错误
- 致命错误

1. 警告并不终止 L51 的执行。这时产生的程序模块由程序员自己斟酌使用还是不使用。但此时的列表文件和屏幕显示可能非常有用。

2. 错误并不终止 L51 的执行。这时产生的程序模块是不能使用的。但此时的列表文件和屏幕显示可能非常有用。

3. 致命错误发生，立即终止 L51 的执行。

下面给出 L51 连接/定位器的所有使用错误信息、原因和解决办法。

### C.2 L51 警告

· WARNING1: UNSOLVED EXTERNAL SYMBOLS

SYMBOLS: external \_ name

MODULE: filename(modulename)

指定模块的外部符号在 PUBLIC 符号表中找不到。

· WARNING2: REFERENCE MADE TO UNSOLVED EXTERNAL

SYMBOLS: external \_ name

MODULE: filename(modulename)

ADDRESS: code \_ address

访问了未能匹配的外部符号 code 地址。

· WARNING3: ASSIGNED ADDRESS NOT COMPITIBLE WITH ALIGNMENT

SEGMENT: segment \_ name

给指定段分配的地址与对齐规则不符。

· WARNING4: DATA SPACE MEMORY OVERLAP

FROM: byte. bit address

TO: byte. bit address

数据空间指定范围出现覆盖。

· WARNING5: CODE SPACE MEMORY OVERLAP

FROM: byte.bit address

TO: byte.bit address

程序空间指定范围出现覆盖。



· WARNING6: XDATA SPACE MEMORY OVERLAP

FROM: byte.bit address

TO: byte.bit address

外部数据空间指定范围出现覆盖。

· WARNING7: MODULE NAME NOT UNIQUE

MODULE: filename(modulename)

模块名重名。模块未处理。

· WARNING8: MODULE NAME EXPLICITLY REQUESTED FROM ANOTHER FILE

MODULE: filename(modulename)

其他文件指名要求本模块名。

· WARNING9: EMPTY ABSOLUTE SEGMENT

MODULE: filename(modulename)

本模块包括空的绝对段,因未定位,它可能在不通知的情况下随时被覆盖。

· WARNING10: CANNOT DETERMINE ROOT SEGMENT

L51 对输入文件要分辨是 C51 文件还是 PL/M 文件,然后进行流程分析,在无法确定根段的时候,发出本警告。它发生在主程序被汇编调用的时候。需要程序员用 OVERLAY 特殊控制选项进行干预。

· WARNING11: CANNOT FIND SEGMENT OR FUNCTION NAME

NAME: overlay \_ control \_ name

在目标模块中找不到 OVERLAY 控制选项中规定的段或函数名。

· WARNING12: NO REFERENCE BETWEEN SEGMENTS

SEGMENT1: segment \_ name

SEGMENT2: segment \_ name

试图用 OVERLAY 控制选项删除本来不存在的段间访问或函数间调用。

· WARNING13: RECURSIVE CALL TO SEGMENT

SEGMENT: segment \_ name

CALLER: segment \_ name

CALLER 段递归调用 SEGMENT 段。PL/M51 和 C51 的非重入函数不允许递归调用。

· WARNING14: IMCOMPITIBLE MEMORY MODEL

MODULE: filename(modulename)

MODEL: memory model

指定模块试图用与以前不同的存储模式编译。

· WARNING15: MULTICALL TO SEGMENT

SEGMENT: segment \_ name

CALLER1: segment \_ name

CALLER2: segment \_ name

两个函数调用同一个函数(如主函数和中断函数),参数和局部变量将被覆盖。

· WARNING16: UNCALLED SEGMENT, IGNORED FOR OVERLAY PROCESS

SEGMENT: segment \_ name

所给段未被调用(可能用于测试),已被排除在覆盖过程之外。调用这个段占用覆盖外的空间。

### C.3 L51 错误

· ERROR 101 SEGMENT COMBINATION ERROR

SEGMENT: segment \_ name

MODULE: filename(modulename)

由于连接错所给段未能连入类型总段,并被忽略。

· ERROR 102 EXTERN ATTRIBUTE MISMATCH

SYMBOL: external \_ name

MODULE: filename(modulename)

所给外部符号名属性错,并被忽略。

· ERROR 103 EXTERN ATTRIBUTE DO NOT MATCH PUBLIC

SYMBOL: public \_ name

MODULE: filename(modulename)

所给外部符号名属性与公用符号名不匹配,并被忽略。

· ERROR 104 MULTI PUBLIC DEFINATIONS

SYMBOL: public \_ name  
MODULE: filename(modulename)  
所给公用符号重名。

• ERROR 105 PUBLIC REFERS TO IGNORED SEGMENT

SYMBOL: public \_ name  
MODULE: filename(modulename)  
所给外部符号名属性错,并被忽略。

• ERROR 106 SEGMENT OVERFLOW

SEGMENT: segment \_ name  
所给段长超过 64K,未处理。

• ERROR 107 ADDRESS SPACE OVERFLOW

SPACE: space \_ name  
SEGMENT: segment \_ name  
由于存储空间不够所给类型总段未能装入,已被忽略。

• ERROR 108 SEGMENT IN LOCATING CONTROL CANNOT ALLOCATED

SEGMENT: segment \_ name  
命令行定位控制中的段由于属性问题未能分配。

• ERROR 109 EMPTY RELOCATABLE SEGMENT

SEGMENT: segment \_ name  
可再定位类型总段长度为零,未定位。

• ERROR 110 CANNOT FIND SEGMENT

SEGMENT: segment \_ name  
命令行所给的段在输入模块中未能找到,被忽略。

• ERROR 111 SPECIFIED BIT ADDRESS NOT ON BYTE MEMORY

SEGMENT: segment \_ name  
位地址不在字节界上,位段被忽略。

• ERROR 112 SEGMENT TYPE NOT LEGAL FOR COMMAND

SEGMENT: segment \_ name  
命令行所给的段类型非法,被忽略。

• ERROR 114 SEGMENT DOES NOT FIT

SPACE: space \_ name  
SEGMENT: segment \_ name  
BASE: base \_ address  
LENGTH: segment \_ length

由于所给段的长度或基地址问题未能定位,并被忽略。

• ERROR 115 INPAGE SEGMENT IS GREATER THAN 256 BYTE

SEGMENT: segment \_ name

所给 INPAGE 属性的段长于 256 字节未能连入类型总段,并被忽略。

• ERROR 116 INBLOCK SEGMENT IS GREATER THAN 2048 BYTES

SEGMENT: segment \_ name

所给 INBLOCK 属性的段长于 2K 字节未能连入类型总段,并被忽略。

• ERROR 117 BITADDRESSABLE SEGMENT IS GREAT THAN 16 BYTES

SEGMENT: segment \_ name

所给 BITADDRESSABLE 属性的段长于 16 字节未能连入类型总段,被忽略。

• ERROR 118 REFERENCE MADE TO ERRONEOUS EXTENAL

SYMBOL symbol \_ name

MODULE: filename(modulename)

ASSRESS: code \_ address

企图访问错误的外部程序地址。

• ERROR 119 REFERENCE MADE TO ERRONEOUS SEGMENT

SEGMENT: segment \_ name

MODULE: filename(modulename)

ASSRESS: code \_ address

企图访问错误段的程序地址。

• ERROR 120 CONTENT BELONGS TO ERRONEOUS SEGMENT

SEGMENT: segment \_ name

MODULE: filename(modulename)

该内容属于有错误的段。

• ERROR 121 IMPROPER FIXUP

MODULE: filename(modulename)

SEGMENT: segment \_ name

超星图书馆  
使用本复制品  
请尊重相关知识产权!

OFFSET:        segment \_ address

根据所给段和偏移地址得到的是不当的地址。

• ERROR 122        CANNOT FIND MODULE

MODULE:        filename(modulename)

命令行所给的模块未能找到。

## C.4 L51 致命错误

• FATAL ERROR201    INVALID COMMAND LINE SYNTAX

partial comand line

命令行句法错。命令行显示到出错处。

• FATAL ERROR202    INVALID COMMAND LINE, TOKEN TOO LONG

partial comand line

非法命令行, 单词太长。命令行显示到出错处。

• FATAL ERROR203    EXPECTED ITEM MISSING

partial comand line

缺项。命令行显示到出错处。

• FATAL ERROR204    INVALID KEYWORD

partial comand line

非法关键字。

• FATAL ERROR205    CONSTANT TOO LONG

partial comand line

常量大于 0xffff。命令行显示到出错处。

• FATAL ERROR206    INVALID CONSTANT

partial comand line

命令行常量无效(如十六进制数以字母开头)。命令行显示到出错处。

• FATAL ERROR207    INVALID NAME

partial comand line

模块或段名无效。命令行显示到出错处。

• FATAL ERROR208    INVALID FILENAME

partial comand line

文件名无效。命令行显示到出错处。

• FATAL ERROR209 FILE USED IN CONFLICTING CONTEXTS

FILE: filename

所给文件名用于有矛盾之处。命令行显示到出错处。

• FATAL ERROR210 I/O ERROR ON INPUT FILE

system error message

FILE: filename

访问输入文件时检测到有错,并由系统 EXCEPTION 给出错误信息。

• FATAL ERROR211 I/O ERROR ON OUTPUT FILE

system error message

FILE: filename

访问输出文件时检测到有错,并由系统 EXCEPTION 给出错误信息。

• FATAL ERROR212 I/O ERROR ON LISTING FILE

system error message

FILE: filename

访问列表文件时检测到有错,并由系统 EXCEPTION 给出错误信息。

• FATAL ERROR213 I/O ERROR ON WORK FILE

system error message

FILE: filename

访问工作文件时检测到有错,并由系统 EXCEPTION 给出错误信息。

• FATAL ERROR214 INPUT PHASE ERROR

MODULE: filename(modulename)

L51 在二趟扫描时发现输入阶段有错,可能因汇编错误引起的。

• FATAL ERROR215 CHECK SUM ERROR

MODULE: filename(modulename)

文件检查和错。

• FATAL ERROR216 INSUFFICIENT MEMORY

执行 L51 内存不够。

• FATAL ERROR217 NO MODULE TO BE PROCESSED

缺少应被处理的模块。





- FATAL ERROR218 NOT AN OBJECT FILE  
FILE: filename  
所给文件非目标文件。
- FATAL ERROR219 NOT AN 8051 OBJECT FILE  
FILE: filename  
所给文件非 8051 目标文件。
- FATAL ERROR220 INVALID INPUT MODULE  
FILE: filename  
所给输入模块无效,可能是因汇编错误引起的。
- FATAL ERROR221 MODULE SPECIFIED MORE THAN ONCE  
partial comand line  
命令行上多次出现所指模块。命令行显示到出错处。
- FATAL ERROR222 SEGMENT SPECIFIED MORE THAN ONCE  
partial comand line  
命令行上多次出现所指的段。命令行显示到出错处。
- FATAL ERROR224 DUPLICATE KEYWORD OR CONFLICTING CON - TROL  
partial comand line  
命令行上多次出现所指的关键字或存在相矛盾的控制选项。命令行显示到出错处。
- FATAL ERROR225 SEGMENT ADDRESS ARE NOT IN ASCENDING ORDER  
partial comand line  
命令行上段地址未按升幂排列。命令行显示到出错处。
- FATAL ERROR226 SEGMENT ADDRESS INVALID FOR CONTROL  
partial comand line  
命令行上段地址无效。命令行显示到出错处。
- FATAL ERROR227 PARAMETER OUT OF RANGE  
partial comand line  
所给 PAGEWIDTH 和 PAGELENGTH 的参数越界。命令行显示到出错处。
- FATAL ERROR228 PARAMETER OUT OF RANGE  
partial comand line  
命令行上 RAMSIZE 的参数越界。命令行显示到出错处。

- FATAL ERROR229 INTERNAL PROCESS ERROR

参数越界 L51 检测到内部处理错。



- FATAL ERROR230 STATRT ADDRESS SPECIFIED MORE THAN ONCE

partial comand line

命令行上起始地址重复。命令行显示到出错处。

- FATAL ERROR233 ILLEGAL USE OF \* IN OVERLAY CONTROL

partial comand line

命令行 OVERLAY 定位选项非法使用了 \* 号(如 \*! \* 或 \* ~ \*)。命令行显示到出错处。

## C.5 例外信息

L51 的某些错误的原因由操作系统的 EXCEPTION 给出。这些信息如下：

- EXCEPTION 0021H PATH OR FILE NOT FOUND

路径名或文件名未找到。

- EXCEPTION 0026H ILLEGAL FILE ACCESS

试图写或删除写保护文件。

- EXCEPTION 0029H ACCESS FILE DENIED

所给文件实际是目录。

- EXCEPTION 002AH I/O - ERROR

欲写的驱动器已满或未准备好。

- EXCEPTION 0101H ILLEGAL CONTEXT

命令的语义非法。如打开打印机读。

## 附录 D C51 的极限值

- 标识符最长 255 个字符,一般取 32 个字符。大小写不敏感。
- case 语句的变量个数没有限制,仅受可用内存容量和函数的最大长度的限制。
- 函数嵌套调用最大深度为 10。
- 功能块{...}最大嵌套深度为 15。
- 宏最多嵌套深度为 8。
- 函数及宏的参数最多 32 个。
- 语句行和宏定义行最多 510 个字符(宏扩展后是 510 个字符)。
- 头文件嵌套深度为 20。
- 预处理器中的条件编译层最多为 20。
- 关于 Intel 目标模块格式(OMF~51)的极限值:
  - 函数类型段总和最多 255 个;
  - 全局符号(PUBLIC)最多 256 个;
  - 外部符号(EXTERNALS)最多 256 个。

## 附录 E XAC 运行时间库函数

下面按字母顺序给出 XAC 运行时间库中的函数说明。

### E.1 ACOS

·提要

```
#include <math.h>
double acos(double f)
```

·说明

给闭区间 $[-1, +1]$ 上的  $f$  值, 返回对应的反余弦( $\text{acos}$ )角的弧度值。

[例 E.1]

```
#include <math.h>
#include <stdio.h>
```

```
main( )
```

```
{
    float i, a;
```

```
    for(i = -1.0; i < 1.0; i += 0.1)
```

```
    {
        a = acod(i) * 180.0/3.141592;
        printf("acos( %f) = %f degrees \n", i, a);
    }
```

·参见

$\sin$ ,  $\cos$ ,  $\tan$ ,  $\text{asin}$ ,  $\text{atan}$ ,  $\text{atan2}$ 。

·返回值

返回  $0 \sim \pi$ , 给值超范围时返回 0。

### E.2 ASCTIME

·提要

```
#include <time.h>
char * asctime(struct tm * t)
```

·说明

将参数所指定的  $\text{tm}$  结构指针指向的  $\text{int}$  细部日期时间转换成如下的 26 个 ASCII 字符组成的全日期时间:

Sun Sep 16 01:02:55 1997 \n \0

函数返回指向上述 ASCII 字符串的字符指针。在下例中先取时间,再转换为 tm 结构的当地时间,最后显示转换为 ASCII 时间。

[例 E.2]

```
#include <time.h>
#include <stdio.h>

main( )
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("%s", asctime(tp));
}
```

·参见

ctime, time, gmtime, localtime, time。

·返回值

返回字符指针指向如下的 26 字符串的 ASCII 全日期时间:

Sun Sep 16 01:02:55 1997 \n \0

·数据类型

```
struct tm
{
    int tm_sec,
        tm_min,
        tm_hour,
        tm_mday,
        tm_mon,
        tm_year,
        tm_wday,
        tm_yday,
        tm_isdat;
}
```

## E.3 ASIN

·提要

```
#include <math.h>
double asin(double f)
```

·说明

给闭区间 $[-1, +1]$ 上的  $f$  值, 返回对应的反正弦( $\text{asin}$ )角的弧度值。

### [例 E.3]

```
#include <math.h>
#include <stdio.h>
```

```
main( )
{
    float f, a;

    for(i = -1.0; i < 1.0; i += 0.1)
    {
        a = asin(i) * 180.0/3.141592;
        printf("asin( %f) = %f degrees \n", i, a);
    }
}
```

#### ·参见

$\sin$ ,  $\cos$ ,  $\tan$ ,  $\text{acos}$ ,  $\text{atan}$ ,  $\text{atan2}$ 。

#### ·返回值

返回  $(-1/2 \sim 1/2) * \pi$ , 给值超范围时返回 0。

## E.4 ASSERT

#### ·提要

```
#include <assert.h>
void assert(int e)
```

#### ·说明

它是用于调试的宏。在程序运行时某些条件确定能成立的点上放本函数, 万一条件不满足, 会给出错信息, 并调用  $\text{abort}()$  终止程序运行。错误信息格式如下:

Assertion failed:  $\text{assert}()$  的实参, 文件名, 错误行号。

### [例 E.4]

```
void printfunc(struct xyz * tp)
{
    assert(tp != 0);
}

int main( )
{
    printfunc(NULL);
    return 0;
}
```

#### 因错输出

Assertion failed:  $\text{tp} != 0$ , file C: \HPDXA\assert.c, line 8

超星阅读器提醒您：  
使用本复制品  
请尊重相关知识产权！

## ·返回值

无。

超星阅读器提醒您：  
使用本复制品  
请尊重相关知识产权！

## E.5 ATAN

## ·提要

```
#include <math.h>
double atan(double f)
```

## ·说明

给开区间  $(-, +)$  上的  $f$  值, 返回对应的反正切( $\text{atan}$ )角的弧度值。

## [例 E.5]

```
#include <math.h>
#include <stdio.h>

main( )
{
    float i, a;

    for(i = -1.0; i < 1.0; i += 0.1)
    {
        a = atan(i) * 180.0 / 3.141592;
        printf("atan( %f) = %f degrees \n", i, a);
    }
}
```

## ·参见

$\sin$ ,  $\cos$ ,  $\tan$ ,  $\text{acos}$ ,  $\text{asin}$ ,  $\text{atan2}$ 。

## ·返回值

返回  $(-1/2 \sim 1/2) * \pi$ 。

## E.6 ATOF

## ·提要

```
#include <stdlib.h>
double atof(char *s)
```

## ·说明

$\text{atof}()$  扫描字符串, 跳过前导空白符后, 将数的每一个字符取出转换为浮点数返回。浮点数可以表示为: 带小数点的十进制数、指数形式或科学指数形式。

## [例 E.6]

```
#include <stdlib.h>
#include <stdio.h>
```



超星阅读器提醒您：  
使用本复制品  
请尊重相关知识产权！

```
main( )  
{  
    char buf[80];  
    double i;  
  
    gets(buf);  
    i = atof(buf);  
    printf("Read %s: convert to %f\n", buf, i);  
}
```

• 参见

atoi, atol。

• 返回值

双精度浮点数。如在字符串中未发现数, 返回 0.0。

## E.7 ATOI

• 提要

```
#include <stdlib.h>  
int atoi(char *s)
```

• 说明

atoi( ) 扫描字符串, 跳过前导空白符将属于数的每一个字符取出转换为整型数返回。

[例 E.7]

```
#include <stdlib.h>  
#include <stdio.h>
```

```
main( )  
{  
    char buf[80];  
    double i;  
  
    gets(buf);  
    i = atoi(buf);  
    printf("Read %s: convert to %d\n", buf, i);  
}
```

• 参见

atof, atol。

• 返回值

整型数。如在字符串中未发现数, 返回 0。

## E.8 ATOL

• 提要

```
#include <stdlib.h>
```

```
longatol(char *s)
```

·说明

atol( ) 扫描字符串, 跳过前导空白符将属于数的每一个字符取出转换为长整型数返回。

[例 E.8]

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
main( )
```

```
char buf[80];
```

```
long i;
```

```
gets(buf);
```

```
i = atol(buf);
```

```
printf("Read %s: convert to %ld\n", buf, i);
```

·参见

atof, atoi。

·返回值

长整型数。如在字符串中未发现数, 返回 0。

## E.9 BSEARCH

·提要

```
#include <stdlib.h>
```

```
void *bsearch(const void *key, const void *base, size_t n_memb,  
              size_t size, int(*compar)(const void *, const void *))
```

·说明

按照给定的关键字在排序的链表中用二分法寻找匹配的链表元素。

[例 E.9]

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
struct value
```

```
{ char name[40];
```

```
  int value;
```

```
} values[100]
```

```
int val_cmpa(const void *p1, const void *p2)
```

```
| return strcmp((const struct value *)p1 ->name, (const struct *)p2 ->name)
```

```
|
|
| main( )
```

```
| char inbuf[80];
```

```
| int i;
```

```
| struct value * vp;
```

```
| i = 0;
```

```
| while(gets(inbuf))
```

```
| | sscanf(inbuf, "%s %d", &values[i].name, &values[i].value);
```

```
| | i++;
```

```
| qsort(values, i, sizeof(values[0]), val_cmp);
```

```
| vp = bsearch("fred", values, i, sizeof(values[0]), val_cmp);
```

```
| if(! vp)
```

```
| | print("item 'fred' was not found. \n");
```

```
| else
```

```
| | printf("item 'fred' has valu %d \n", vp ->value);
```

```
|
|
| • 参见
```

```
| | qsort( )。
```

```
|
| • 返回值
```

指向匹配数组元素的指针。无匹配返回空指针。

## E.10 CALLOC

• 提要

```
#include <stdlib.h>
```

```
void * calloc(size_t cnt, size_t size)
```

• 说明

给指定个数、指定大小的对象分配动态存储块,并将块清 0。分配成功返回指向存储块的指针,失败返回空指针。

[例 E.10]

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
struct test
```

```
|
```

```
| int a[20];
```

| \* ptr

main( )

```
ptr = calloc(20, sizeof(struct test));
if(! ptr)
    printf("Failed \n");
else
    free(ptr);
```

·参见

brk, sbrk, malloc, free。

·返回值

分配成功返回指向存储块的指针, 失败返回空指针。

## E.11 CEIL

·提要

```
#include <math.h>
double ceil(double f)
```

·说明

返回不小于  $f$  的只有整数的 double 数。

[例 E.11]

```
#include <math.h>
#include <stdio.h>
main( )
{ float x, y;
  x = 12.3;
  y = ceil(x);
  printf("ceil (%f) = %f \n", x, y);
}
```

·参见

floor。

·返回值

有效整数部分的 double 数。

## E.12 CGETS

·提要

```
#include <conio.h>
char * cgets(char * s)
```

## ·说明

由控制台读入一行字符,放指定缓冲器中。它重复调用 `getche()`, 因有缓冲器,可用 BACKSPACE 消光标前字符,或用 `ctrl-U` 消光标前全行。行的最大字符数、回车和换行符将结束本函数的执行。返回与参数一样的指针。

## [例 E.12]

```
#include <conio.h>

char buf[80];

main( )
{
    for(;;)
    {
        cgets(buf);
        if(strcmp(buf, "exit") == 0)
            break;
        cputs("Type 'exit' to finish \n");
    }
}
```

## ·参见

`getch`, `getche`, `putch`, `cputs`。

## ·返回值

返回与参数一样的指针。

## E.13 COS

## ·提要

```
#include <math.h>
double cos(double f)
```

## ·说明

求指定参数的余弦值。用多项式展开求近似值。

## [例 E.13]

```
#include <math.h>
#include <stdio.h>

#define C3.141592

main( )
{
    double i;

    for(i=0; i<180.0; i+=10)
```

```
printf("sin( %3.0f) = %f, cos( %3.0f) = %f \n", i, sin(i * C/180.0), i, cos(i * C/180.0));
```

·参见

sin, tan, asin, acos, atan, atan2。

·返回值

-1.0~1.0 范围的 double 数。

## E.14 COSH, SINH, TANH

·提要

```
#include <math.h>
double cosh(double f)
double sinh(double f)
double tanh(double f)
```

·说明

它们求解双曲线三角函数。

## E.15 CPUTS

·提要

```
#include <conio.h>
void cputs(char * s)
```

·说明

将参数指定字符串写到控制台。它重复调用 `putch()`。换行符前要加回车符。换行符或字符串终结符结束本函数的执行。在系统机上 `cputs()` 与 `puts()` 是有区别的, `cputs()` 是直接 from 控制台上读入的。在嵌入式系统中, 二者是等价的。

[例 E.15]

```
#include <conio.h>
```

```
char buf[80];
```

```
main( )
```

```
{
```

```
for(;;)
```

```
{
```

```
    cgets(buf);
```

```
    if(strcmp(buf, "exit") == 0)
```

```
        break;
```

```
    cputs("Type 'exit' to finish \n");
```

超星阅读器扫描  
使用本复制品  
请尊重相关知识产权!

• 参见

getch, getche, cgets, putch, cputs。

• 返回值

无。

## E.16 CTIME

• 提要

```
#include <time.h>
```

```
char * ctime(time_t t)
```

• 说明

用参数所给的秒时间转换成 26 字符串的格式化的当前全日期时间:

```
Sun Sep 16 01:02:55 1997 \n \0
```

返回字符指针指向此 26 字符串。例中先取秒时间,再显示转换为 ASCII 的时间。

[例 E.16]

```
#include <time.h>
```

```
#include <stdio.h>
```

```
main( )
```

```
time_t clock;
```

```
time(&clock);
```

```
printf("%s", ctime(&clock));
```

• 参见

asctime, time, gmtime, localtime, time。

• 返回值

返回字符指针指向如下的 26 字符串的格式化的全日期时间:

```
Sun Sep 16 01:02:55 1997 \n \0
```

• 数据类型

```
typedef long time_t;
```

## E.17 DI, EI

• 提要

```
#include <intrpt.h>
```

```
void di(void)
```



```
void ei(void)
```

·说明

ei() 和 di() 分别开中断和关中断。它们在 intrpt.h 中用宏写成, 展开时是直接的在线汇编码。

[例 E.17]

```
#include <intrpt.h>
```

```
long count;
```

```
void interrupttick(void)
```

```
{ count ++;
```

```
}
```

```
long getticks(void)
```

```
{ longval;
```

```
di();
```

```
val = count;
```

```
ei();
```

```
return(val);
```

```
}
```

## E.18 DIV

·提要

```
#include <stdlib.h>
```

```
div_t div(int numer, int denom)
```

·说明

求 numer 除以 denomsuo 所得的商和余。返回由商和余定义的结构类型 div\_t。

[例 E.18]

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
main()
```

```
{ div_t t;
```

```
t = div(1234567, 12345);
```

```
printf("Quotient = %d, remainder = %d \n", t.quot, t.rem);
```

```
}
```

·参见

ldiv

- 返回值

类型为 `div_t` 的结构。

返回值为 0。



```
main(·)
{ double f;

  for (f=0.0;f<=5;f+=1.0)
    printf("e to %1.0f = %f\n",f,exp(f));
}
```

·参见

log, log10, pow。

·返回值

无。

超星阅读器提醒您：  
使用本复制品  
请尊重相关知识产权！

## E.21 fabs

·提要

```
#include <math.h>
```

```
double fabs(double f)
```

·说明

返回 double 参数的绝对值。

[例 E.20]

```
#include <math.h>
```

```
#include <stdio.h>
```

```
main(void)
```

```
{ float x,y;
```

```
  x = -3.4;
```

```
  y = fabs(x);
```

```
  printf("fabs(%f) = %f\n",x,y);
```

·参见

abs。

·返回值

绝对值。

## E.22 FLOOR

·提要

```
#include <math.h>
```

```
double floor(double f)
```

·说明

返回不大 f 的最大整数。

[例 E.22]

```
#include <math.h>
#include <stdio.h>

main( )
{
    float x, y;
    x = 12.3;
    y = f(x);
    printf("floor( %f) = %f \n", x, y);
}
```

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

• 参见

ceil。

• 返回值

只有整数部分的 double 数。

## E.23 FREE

• 提要

```
#include <stdlib.h>
```

```
void free(void * ptr)
```

• 说明

释放由 calloc( )和 malloc( )分配的 ptr 指向的存储块。

[例 E.22]

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
struct test
```

```
{
```

```
    inta[20];
```

```
| * ptr
```

```
    main( )
```

```
{
```

```
    ptr = calloc(20, sizeof(struct test));
```

```
    if(! ptr)
```

```
        printf("Failed \n");
```

```
    else
```

```
        free(ptr);
```

```
|
```

• 参见

malloc, calloc。

• 返回值

无。

## E.24 FREXP

### ·提要

```
#include <math.h>
double frexp(double f, int *p)
```

### ·说明

将浮点数分为规范化小数部分和 2 的整数幂部分。整数幂放在 p 指向的 int 对象中。返回值是规范化小数部分。规范化小数的取值范围为[0.5, 1.0)和 0。

### [例 E.24]

```
#include <stdio.h>
#include <math.h>
```

```
main( )
{
    double f;
    int i;

    f = frexp(23456.78, &i);
    printf("23456.78 = %f * 2^%d\n", f, i);
}
```

### ·参见

ldexp。

### ·返回值

规范化小数部分。

## E.25 GETC

### ·提要

```
#include <stdio.h>
FILE * stream;
int getc(FILE * stream)
```

### ·说明

由参数指定的流中读一个字符并返回它。它是宏版本的 fgetc()。头文件是 stdio.h。

### [例 E.25]

```
#include <stdio.h>
```

```
main( )
{
    int i;
```



```
while((i = getc(stdin)) != EOF)
    putchar(i);
```

• 参见

getch, getche, cgets, putch, cputs。

• 返回值

文件读完返回 EOF, 中间有错返回 EOF。



## E.26 GETCH, GETCHE, UNGETCH

• 提要

```
#include <conio.h>
char getch(void)
char getche(void)
void getch(char)
```

• 说明

getch( ) 由控制台键盘读一个字符并返回它。getche( ) 同前但同时送标准输出。ungetch( ) 退回一个字符以便下次再读, 对于控制台屏幕来说, 遇到新行退回等待补回车符。对于嵌入系统, 输入源由特殊的程序供给。作为缺省, 库中提供一个接口到 LUCIFER 调试器的 getch( ) 版本。用户可根据自己的需要提供字符的输入源, 如串行口等。

[例 E.26]

```
#include <conio.h>

main( )
{
    chari;

    while((i = getche()) != '\n');
```

• 参见

cgets, cputs。

• 返回值

getch(void), getche(void) 返回字符, ungetch( ) 无返回。

## E.27 GETS

• 提要

```
#include <stdio.h>

char * gets(char * s)
```

## ·说明

gets( ) 由标准输入读一行到缓冲器删除换行符, 加上 null 终结符。对于嵌入系统, gets( ) 与 cgets( ) 等价。gets( ) 重复调用 getche( ), 可以使用退格键等删除字符。

## [例 E.27]

```
#include <stdio.h>
```

```
main( )
```

```
{
```

```
    charbuf[80];
```

```
    printf("Type a line:");
```

```
    if(gets(buf))
```

```
        puts(buf);
```

```
}
```

## ·参见

fgets, freopen, puts。

## ·返回值

一行结束返回 null, 中间返回同与参数。

## E.28 GMTIME

## ·提要

```
#include <time.h>
```

```
struct tm * gmtime(time_t * t)
```

## ·说明

将参数指针获取的当前累计秒数转换成为细部时间放结构 tm 中, 返回 structtm 型结构。

## [例 E.28]

```
#include <time.h>
```

```
#include <stdio.h>
```

```
main( )
```

```
{
```

```
    time_t clock;
```

```
    struct tm * tp;
```

```
    time(&clock);
```

```
    tp = gmtime(&clock);
```

```
    printf("It's %d in London \n", tp->year + 1900);
```

```
}
```

## ·参见

ctime, time, asctime, localtime。





## ·返回值

返回结构 tm。

## ·数据类型

```
struct tm
{
    int tm_sec,
        tm_min,
        tm_hour,
        tm_mday,
        tm_mon,
        tm_year,    /* 年号-1900 */
        tm_wday,    /* 使用夏时制不为 0 */
        tm_yday,
        tm_isdat;
```

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

## E.29 ISALNUM, ISALPHA, ISDIGIT, ISLOWER 等

## ·提要

```
#include <ctype.h>
int isalnum(char c)
int isalpha(char c)
int isascii(char c)
int iscntrl(char c)
int isdigit(char c)
int islower(char c)
int isprint(char c)
int isgraph(char c)
int ispunct(char c)
int isspace(char c)
int isupper(char c)
```

## ·说明

只要字符 c 通过 isascii( ) 或 c = EOF 的检查, 就可以用上述函数进行归类。它们实际上都是宏。

## [例 E.29]

```
#include <ctype.h>
#include <stdio.h>
```

```
main( )
```

```
char buf[80];
```

```
int i;

gets(buf);
i=0;
while(isalnum(buf[i++])) ;
buf[i]=0;
printf("%s is the buf\n", buf);
}
```

• 参见

toupper, tolower, toascii。

• 返回值

测试为真返回非 0 值。

## E.30 KBHIT

• 提要

```
#include <conio.h>
```

```
int kbhit(void)
```

• 说明

测试控制台键盘是否有键按下, 真返回非 0 值; 否返回 0 值。一般随后用 getch() 读字符。

[例 E.30]

```
#include <conio.h>
```

```
main( )
```

```
{
int i;
```

```
while(! kbhit( ))
```

```
{ cputs("I'm waiting");
```

```
for (i=0; i!=100; i++)
```

```
continue;
```

• 参见

getch, getche。

• 返回值

真返回非 0 值; 否返回 0 值。

## E.31 LDEXP

• 提要



```
#include <math.h>
```

```
double ldexp(double f, int *p)
```

•说明

是 frexp() 的逆。将浮点数的规范化小数部分和 2 的整数幂部分合并为带小数点的浮点数。

[例 E.31]

```
#include <stdio.h>
```

```
#include <math.h>
```

```
main( )
```

```
{ double f;
```

```
    f = ldexp(1.0, 10);
```

```
    printf("1.0 * 2^10 = %f \n", f);
```

•参见

frexp。

•返回值

规范化小数部分。

## E.32 LDIV

•提要

```
#include <stdlib.h>
```

```
ldiv_t ldiv(long numer, long denom)
```

•说明

求 numer 除以 denom 所得的商和余。返回由商和余构成的 ldiv\_t 类型结构。

[例 E.32]

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
main( )
```

```
{ ldiv_t lt;
```

```
    lt = ldiv(1234567, 12345);
```

```
    printf("Quotient = %ld, remainder = %ld \n", lt.quot, lt.rem);
```

•参见

div。

•返回值

类型为 ldiv\_t 的结构。

## •数据类型

```
typedef struct
{
    long quot;
    long rem;
    ldiv_t;
```

超星浏览器提醒您：  
使用本复制品  
请尊重相关知识产权！

## E.33 localtime

## •提要

```
#include <time.h>
struct tm * localtime(time_t * t)
```

## •说明

将参数指针获取的当前累计秒数转换并修正成为细部时间,放结构 tm 中返回。

## [例 E.33]

```
#include <time.h>
#include <stdio.h>

char * wday[] =
{ "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };

main( )
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("Today is %s\n", wday[tp->tm_wdy]);
}
```

## •参见

ctime, time, asctime。

## •返回值

返回结构 tm。

## •数据类型

```
struct tm
{
    int tm_sec,
        tm_min,
        tm_hour,
        tm_mday,
        tm_mon,
        tm_year,
```

```
tm_wday,
tm_yday,
tm_isdat;
```



## E.34 LOG, LOG10

### ·提要

```
#include <math.h>
double log(double f)
double log10(double f)
```

### ·说明

它们分别求参数  $f$  的自然对数和以 10 为底的对数。

### [例 E.34]

```
#include <stdio.h>
#include <stdlib.h>
```

```
main( )
{ double f;

  for (f=1.0;f<=10.0;f+=1.0)
    printf("log( %1.0f) = %f\n",f,log(f));
}
```

### ·参见

exp, pow。

### ·返回值

返回对数值, 参数为负, 返回 0。

## E.35 LONGJMP

### ·提要

```
#include <setjmp.h>
void longjmp(jmp_buf buf, int val)
```

### ·说明

longjmp( ) 和 setjmp( ) 联合实现非局部 goto。在外层函数初次调用带 jmp\_buf 参数的 setjmp( ) 时, 返回的是 0 值。内层函数通过调用 longjmp( ), 实现非局部 goto 返回到外层函数。longjmp( ) 同样以 jmp\_buf 为参数, 与此同时还给一个非 0 值的 val 参数, 使在 setjmp( ) 有效期内, 再次调用 setjmp( ) 时, 返回的是非 0 的 val 值, 以区别是由何处调用的 setjmp( )。外层函数初次调用 setjmp( ) 后即处于等待由内层返回的有效期间。违背在 setjmp( ) 有效期内调用 longjmp( ), 将产生灾难性的后果, 因为堆栈数据已变。

## [例 E.35]

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf jb;

int inner(void)
{
    longjmp(jm, 5);
}

main(.)
{
    int i;

    if(i = setjmp(jb))
    { printf("longjmp returned %d \n");
      exit(0);
    }

    printf("setjmp returned 0 - good \n");
    printf("about calling inner... \n");
    inner( );
    printf("inner returned - bad \n");
}
```



·参见

setjmp。

·返回值

无。

## E.36 MALLOC

·提要

```
#include <stdlib.h>
void * malloc(size_t cnt)
```

·说明

从堆中动态分配 cnt 字节的存储块，分配成功返回指向存储块的指针，失败返回空指针。  
malloc( ) 调 sbrk( ) 获得存储块。存储块未被清 0。

## [例 E.36]

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
```

```
main( )
```

```
{
char *cp;
```

```
cp = malloc(80);
```

```
if(! cp)
```

```
printf("Malloc Failed \n");
```

```
else
```

```
{ strcpy(cp, "a string");
```

```
printf("block = %s \n", cp);
```

```
free(cp);
```

```
}
```

• 参见

brk, sbrk, calloc, realloc, free。

• 返回值

分配成功返回指向存储块的指针;失败返回空指针。

## E.38 MEMCHR

• 提要

```
#include <string.h>
```

```
void *memchr(const void *block, int val, size_t length)
```

• 说明

在 block 指向的长为 length 的存储块中搜索 val 值。返回第一次搜索到 val 值的指针。

[例 E.38]

```
#include <stdio.h>
```

```
#include <string.h>
```

```
unsigned int ary[] = {1, 5, 0x6789, 0x23};
```

```
main( )
```

```
{
```

```
char *cp;
```

```
cp = memchr(ary, 0x89, sizeof(ary));
```

```
if(! cp)
```

```
printf("Not found \n");
```





else

```
printf("Found at offset %u\n", cp - (char *)ary);
```

·参见

strchr。

·返回值

返回第一次搜索到 val 值的指针。未搜索到 val 值返回 null 指针。



## E.38 MEMCMP

·提要

```
#include <string.h>
```

```
int memcmp(const void *s1, const void *s2, size_t n)
```

·说明

比较两个长度为  $n$  的存储块, 比较是建立在纯 ASCII 字符序列的基础上进行的。如果块中存在非 ASCII 字符时, 返回值是不确定的。比较相等总是可靠的。相等返回 0,  $s_1$  指向的字符序列小于  $s_2$  时返回 -1, 大于  $s_2$  时返回 1。

[例 E.38]

```
#include <stdio.h>
```

```
#include <string.h>
```

```
main( )
```

```
{
    int buf[10], cow[10], i;
    buf[0] = 1;
    buf[0] = 2;
    buf[0] = 3;
    cow[0] = 1
    cow[0] = 3
    cow[0] = 2
    i = memcmp(buf, cow, 3 * sizeof(int));
    if(i < 0)
        printf("less than\n");
    elseif(i > 0)
        printf("great than\n");
    else
        printf("equal\n");
}
```

·参见

strncpy, strncmp, strchr, memchr, memset。

## • 返回值

相等返回 0,  $s_1$  指向的字符序列小于  $s_2$  的返回 -1, 大于  $s_2$  的返回 1。

## E.39 MEMCPY

## • 提要

```
#include <string.h>
```

```
void *memcpy(void *d, void *s, size_t n)
```



## • 说明

将长度为  $n$  的  $s$  所指向的存储块复制到  $d$ , 复制时未考虑可能发生的覆盖, 返回第一个参数指针  $\text{void } *d$ 。

## [例 E.39]

```
#include <stdio.h>
```

```
#include <string.h>
```

```
main( )
```

```
{  
    char buf[80];
```

```
    memset(buf, 0, sizeof(buf));
```

```
    memcpy(buf, "a partial string", 10);
```

```
    printf("buf = %s\n", buf);  
}
```

## • 参见

strncpy, strncmp, strchr, memchr, memset。

## • 返回值

返回第一个参数指针  $\text{void } *d$ 。

## E.40 MEMMOV

## • 提要

```
#include <string.h>
```

```
void *memmov(void *d, void *s, size_t n)
```

## • 说明

将长度为  $n$  的存储块由  $d$  移位到  $s$ 。它相当于 `memcpy()`, 但考虑了覆盖, 能够自动选择向后还是向前复制, 保证复制的正确性。返回第一个参数指针  $\text{void } *d$ 。

## • 参见

strncpy, strncmp, strchr。

## ·返回值

返回第一个参数指针 void \*d。

## E.41 MEMSET

## ·提要

```
#include <string.h>
void memset(void *s, char c, size_t n)
```

## ·说明

将长度为  $n$  的  $s$  所指向的存储块置为  $c$ 。

## [例 E.40]

```
#include <stdio.h>
#include <string.h>
```

```
main( )
```

```
{
    char buf[80];
```

```
    memcpy(buf, "a partial string", 10);
    memset(buf, '5', sizeof(buf));
    printf("buf = %s\n", buf);
}
```

## ·参见

strncpy, strncmp, strchr, memchr, memset, memcpy。

## ·返回值

无。

## E.42 PERSIST\_CHECK, PERSIST\_VALIDATE

## ·提要

```
#include <sys.h>
int persist_check(void *s, char c, size_t n)
void persist_validate(void)
```

## ·说明

它们用于测试 nvram 存储区的正确性。第一次使用 persist\_validate() 将密码置于隐蔽的变量中, 并求校验和。用 persist\_check() 测试 nvram 存储区的正确性。如果密码和校验和有一个不对, 重新调用 persist\_validate() 再置密码并求校验和, 再用 persist\_check() 测试。测试时应将 persist\_check() 的参数标志置 1, 并配合复位。

## [例 E.41]

超星阅读器提醒您:  
使用本复制品  
请尊重相关知识产权!

```
#include <stdio.h>
#include <sys.h>

persistent long reset_count;

main( )
{
    if(! persist_check(1))
        printf("reset count invalid - zeroed \n");
    else
        printf("reset number %ld \n", reset_count);
    reset_count ++ ; persist_validate( );
    for(;;); /* 等待复位 */
}
```

• 参见

无。

• 返回值

正确 persist\_check( ) 返回非 0, 否则返回 0。

## E.43 POW

• 提要

```
#include <math.h>
double pow(double f, double p)
```

• 说明

求  $f$  的  $p$  次方幂。

[例 E.42]

```
#include <stdio.h>
#include <math.h>

main( )
{
    double f;

    for(f=1.0; f<=10.0; f+=1.0)
        printf("pow(2, %1.0f) \n", pow(2, f));
}
```

• 参见

log, log10, exp

• 返回值



double 方幂值。

## E.44 PRINTF, VPRINTF

### ·提要

```
#include <stdio.h>
int printf(char *fmt, ...)
int vprintf(char *, va_list ap)
```

### ·说明

printf( )按照参数表中给出格式串,将后跟 0~n 个逗号分隔的变量进行转换,转换结果送标准输出设备。变量的个数对应于格式串中以 % 打头的变量转换参数的个数。格式串中的变量转换参数的格式如下(方括号内的域是可选的):

%[flags][width][.precision] type

其中:

- [flags]——标志,可有可无。内容:
  - 输出结果左对齐。
  - + 输出符号数时前面加 + / - 号。
  - '' 输出 + / - 号时 + 号以空格代替。
  - # 与 0, x(X) 连用时,数字以 0, 0x(0X) 为前导。
  - 与 f, g(G), e(E) 连用时,结果中一定包括小数点。
  - b, B 与 d, i, u, o, x(X) 连用时,变量改为 8 位数,如 %bu。
  - l, L 与 d, i, u, o, x(X) 连用时,变量改为 32 位数,如 %lu。
  - \*
  - 抑制下一个变量不输出。
- [width]——指定欲显示的字符串最小宽度。可有可无。
  - n 十进制正整数。如果实际字符数小于 n,左端补空格。
  - 0n 如果实际字符数小于 n,左端补 0。
  - \*
  - 在参数表中指定宽度。
- [.precision]——精度指定符,可有可无。
  - (无) 对 d, i, u, o, x(X) 为 1。
  - 对 f, e(E) 为 6。
  - 对 g(G) 为所有有效数字。
  - 对 s, c 为一字符。
  - .0 对 d, i, u, o, x(X) 精度等同于缺省。
  - 对 f, g(G), e(E) 不输出小数部分。
  - .n 精度要求输出 n 个十进制位或 n 个字符,不足添左零。
  - 超出舍入或丢弃。
  - .\*
  - 在参数表中指定精度。
- type——变量类型



d, i	int(十进制符号数)。
u	unsigned int(十进制无符号数)。
o	int(八进制无符号数)。
x, X	int(十六进制无符号数)。x 时, 用 abcdef; X 时, 用 ABCDEF。
f	float([-]ddd.ddd 的符号数)。
e, E	float([-]d.ddd e[+/-]ddd 的符号数, [-]d.ddd E[+/-]ddd 的符号数)。
g, G	float(在 f, e 中选最适合给定值的形式)。
c	char(单个字符)。
s	string pointera(ASCII 字符串)。
p	pointer。

vprintf( ) 与 printf( ) 基本相同, 但是, 须用变指针参数表代替变量参数表。请见 va\_start( )。

#### [例 E.43]

```
printf("total = %4d%%", 22);
```

显示: total = 22 %

```
printf("size is 0x%lX", size);
```

当 size = 123456, 显示: size is 0x1E240

```
printf("name = %.8s", "a1234567890");
```

显示: name = a1234567

```
printf("xx % *d", 3, 4);
```

显示: xx 4

```
/* vprintf( ) 例 */
```

```
#include <stdio.h>
```

```
int error(char *s, ...)
```

```
{ va_list ap;
```

```
va_start(ap, s);
```

```
printf("error:");
```

```
vprintf(s, ap);
```

```
putchar("\n");
```

```
va_end(ap);
```

```
}
```

```
main( )
```

```
{
```

```
int i;
```

```
i = 3;
```

```
error("tesdting 1 2 %d", i);
```

```
}
```

## ·参见

fprintf, sprintf。

## ·返回值

返回写到标准输出的字符数。

超星阅读器提醒您：  
使用本复制品  
请尊重相关知识产权！

## E.45 PUTCH

## ·提要

```
#include <conio.h>
```

```
void putch(int c)
```

## ·说明

输出字符 c 到控制台屏幕,遇换行符先补回车。在 DOS 下,使用系统 I/O 调用,在嵌入系统中,putch( ) 与硬件有关。在嵌入系统中标准 putch( ) 或是与串行口接口或是与调试器 LUCIFER 接口。

## [例 E.44]

```
#include <conio.h>
```

```
char * x = "This is a string";
```

```
main( )
```

```
{
```

```
char * cp;
```

```
cp = x;
```

```
while( * x)
```

```
    putch( * x++ );
```

```
putch( '\n' );
```

```
}
```

## ·参见

cgets, cputs, getch, getche。

## ·返回值

无。

## E.46 PUTS

## ·提要

```
#include <stdio.h>
```

```
int puts(char * s)
```



## ·说明

输出字符串 *s* 到控制台屏幕, 串尾符不输出但附加换行符。

## [例 E.45]

```
#include <stdio.h>
```

```
main( )
```

```
{
    puts("hello world!");
}
```

## ·参见

cgets, cputs, gets, fputs, freopen, fclose。

## ·返回值

正常返回 0, 有错返回 EOF。

## E.47 QSORT

## ·提要

```
#include <stdlib.h>
```

```
void qsort(void * base, size_t nel, size_t width, int (* func)(void *, void * ))
```

## ·说明

用快速算法对元素长度为 *width*、个数为 *nel*、地址指针为 *base* 的数组进行排序。*func* 是用于快速排序的比较函数的入口地址。比较函数 *func()* 有返回值: 若第一个参数大于第二个返回正数; 等于返回 0; 小于返回负数。*qsort()* 没有返回值。

## [例 E.46]

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int array[] = {567, 23, 456, 1024, 17, 567, 66};
```

```
int sortem(const void * p1, const void * p2)
```

```
{
    return( *(int *)p1 - *(int *)p2);
}
```

```
main( )
```

```
{
```

```
    reg int i;
```

```
    qsort(array, sizeof(array)/sizeof(array[0]), sizeof(array[0]), sortem);
```

```
    for(i=0; i! = sizeof(array)/sizeof(array[0]); i++)
```

```
        printf("%d\t", array[i])
```

```
    putchar( '\n' );
```

·参见

无。

·返回值

qsort( ) 没有返回值。

## E.48 RAND

·提要

```
#include <stdlib.h>
```

```
int rand(void)
```

·说明

rand( ) 是伪随机数发生器。每次调用返回一个范围在 0~32 767 的随机数。由同一初始数产生的随机数是确定的。初始数用 srand( ) 产生。

[例 E.47]

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <time.h>
```

```
main( )
```

```
int i;
```

```
time_t toc;
```

```
time( &toc );
```

```
srand( (int)toc );
```

```
for( i=0; i!=10; i++ )
```

```
    printf( "%d\t", rand( ) );
```

```
    putchar( '\n' );
```

·参见

srand。

·返回值

返回随机数。

## E.49 REALLOC

·提要

```
#include <stdlib.h>
```

```
void * realloc(void * ptr, size_t cnt)
```

#### ·说明

为已分配过的动态存储块进行再分配。方法是：把 ptr 指向的原存储块释放；按现在的 cnt 重新申请动态存储块，分配成功，将原块内容复制到新块；最后，返回新块指针。缩小存储块总是可以成功的。扩大则两可。

#### [例 E.48]

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
main( )
```

```
char * cp;
```

```
cp = malloc(80);
```

```
if(gets(cp))
```

```
    cp = realloc(cp, strlen(cp) + 1);
```

```
printf("buf now %d bytes long\n", strlen(cp) + 1);
```

#### ·参见

malloc, calloc, free。

#### ·返回值

分配成功返回指向存储块的指针，失败返回空指针。

## E.50 SCANF, VSCANF

#### ·提要

```
#include <stdio.h>
int scanf(char * fmt, ...)
int vscanf(char *, va_list ap)
```

#### ·说明

按格式串从标准输入流中提取字符并转换之，转换所得的结果放在后跟的 0~n 个逗号分隔的变量指针指向的地方。变量指针的个数对应于格式化串中以 % 打头的变量转换参数的个数。严格禁止输入变量指针的数目偏少，这时会引发灾难性后果。从标准输入流中提取格式串所须的字符时允许有先导的空格符或制表符，但是它们被读出后随即抛掉。变量转换参数的格式如下(方括号内的域是可选的)：

```
%[flags][width] type
```

其中：

1. [flags]——标志，可有可无。内容：

- b, h 与 d, i, u, o, x(X)连用时, 指针改为字符指针, 如 %bu。  
 l 与 d, i, u, o, x(X)连用时, 指针改为长指针, 如 %lu。  
 \* 抑制本输入项。

2. [width]——指定由控制台输入最多字符数。在读入给定字符数之前遇到空白符或不可转换字符时, 即行停止。

n 十进制正整数。如果实际字符数小于  $n$ , 左端补空格。

3. type——将读入的字符数转换成的变量类型。

- d, i int(十进制符号数)。  
 u unsigned int(十进制无符号数)。  
 o int(八进制无符号数)。  
 x, X int(十六进制无符号数)。  
 f, e, E, g, G float(浮点数)。  
 c char(单个字符)。  
 s string pointer(ASCII 字符串)。

vscanf( )与 scanf( )相似, 但是使用的是变参数表而不是变量指针表。请见 va\_list( )。

[例 E.49]

```
scanf("%d %s", &a, &s);
```

输入: 12s

结果: 12 放 a 中, 's' 放 s 中。

```
scanf("%3c %lf", &c, &f);
```

输入: abcdj -12.66

结果: abc 放 c 中, -12.66 放 f 中。

·参见

fscanf, sscanf, va\_arg。

·返回值

提取及转换成功返回转换成功的参数个数。不成功返回 EOF。

## E.51 SET\_VECTOR

·提要

```
#include <intrpt.h>
```

```
typedef interrupt void( * isr)()
```

```
isr set_vector(isr * vector, isr func)
```

·说明

set\_vector( )初始化中断向量。isr 是自定义类型: 中断函数指针类型。set\_vector( )的第一个参数是放中断函数指针的指针, 即 vector 是放中断函数指针的地址。第二个参数是中断函数指针, 调用的时候用的是中断函数的入口地址, 即函数名。

[例 E.50]

超星图书馆  
使用本复制品  
请尊重相关知识产权!

```
#include <stdlib.h>
#include <signal.h>
#include <intrpt.h>

static far interrupt void brkintr(void)
{
    exit(-1);
}

#define BRKINT 0x23 /* ctrl-break interrupt */
#define BRKINTV ((far isr*)(BRKINT*4))

voidset _ trap(void)
{
    set _ vector(BRKINTV, brkintr);
}
```

#### • 参见

di( ), ei( ), ROM\_VECTOR, RAM\_VECTOR, CHANGE\_VECTOR。

#### • 返回值

返回原中断向量内容,即原中断函数入口地址。set \_ vector 的宏版本没有返回值。

## E.52 SETJMP

#### • 提要

```
#include <setjmp.h>
void setjmp(jmp _ buf buf)
```

#### • 说明

longjmp( )和 setjmp( )联合实现非局部 goto。在外层函数初次调用带 jmp \_ buf 参数的 setjmp( ), 返回的是 0 值。内层函数通过调用 longjmp( ), 实现非局部 goto 返回到外层函数。longjmp( )同样以 jmp \_ buf 为参数,与此同时还给一个非 0 值的 val 参数,使在 setjmp( )有效期内,再次调用 setjmp( )时,返回的是非 0 的 val 值,以区别是由何处调用的 setjmp( )。外层函数初次调用 setjmp( )后即处于等待由内层返回的有效期间。违背在 setjmp( )有效期内调用 longjmp( ), 将产生灾难性的后果,因为堆栈数据已变。

#### [例 E.51]

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
```

```
jmp _ bufjb;
```

```
int inner(void)
```



## ·参见

cos, tan, asin, acos, atan, atan2。

## ·返回值

-1.0~1.0 范围的 double 数。

## E.54 SPRINTF, VSPRINTF

## ·提要

```
#include <stdio.h>
```

```
int sprintf(char * buf, char * fmt, ...)
```

```
int vsprintf(char * buf, char * fmt, va_list ap)
```

## ·说明

sprintf() 将格式化输出送缓冲器 buf。在其函数的参数表中给出格式串, 后跟 0~n 个逗号分隔的变量。变量的个数对应于格式串中以 % 打头的变量转换参数的个数。格式串中变量转换参数的格式如下(方括号内的域是可选的):

%[flags][width][.precision] type

其中: ①[flags] ②[width] ③[.precision] ④type 的解释和内容同 E.44。

vsprintf() 与 sprintf() 基本相同, 但是, 要用变指针参数表代替变量参数表。请见 va\_start()。

## [例 E.53]

```
#include <math.h>
main( )
{ char buf[100];
  int n;
  int a, b;
  float pi;
  a = 123;
  b = 456;
  pi = 3.141 59;
  n = sprintf(buf, "%f\n", 1.1);
  nt = sprintf(buf + n, "%d\n", a);
  nt = sprintf(buf + n, "%d %s %g\n", b, "...", pi);
  printf(buf)
}
```

## ·参见

fprintf, printf, sscanf。

## ·返回值

返回写到缓冲器的字符数。

## E.55 SQRT

## ·提要





```
#include <math.h>
```

```
double sqrt(double f)
```

•说明

求指定参数的平方根。用牛顿法求近似值。

[例 E.54]

```
#include <math.h>
```

```
#include <stdio.h>
```

```
main( )
```

```
{
```

```
double i;
```

```
for(i=0;i<=20.0;i+=1.0)
```

```
printf("sqrt of %.1f= %f\n",i,sqrt(i));
```

```
}
```

•参见

exp。

•返回值

返回平方根。

## E.56 SRAND

•提要

```
#include <stdlib.h>
```

```
void srand(int seed)
```

•说明

srand( )初始化伪随机数发生器。

[例 E.55]

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <time.h>
```

```
main( )
```

```
{
```

```
int i; time_t toc;
```

```
time(&toc);
```

```
srand((int)toc);
```

```
for(i=0;i!=10;i++)
```

```
printf("%d\t",rand());
```

```
putchar('\n');
```

·参见

rand。

·返回值

无。

## E.57 SSCANF, VSSCANF

·提要

```
#include <stdio.h>
```

```
int sscanf(char * buf, char * fmt, ...)
```

```
int vsscanf(char * buf, char * fmt, va_list ap)
```

·说明

从缓冲器中按格式串提取字符并转换之。参数表中给出格式串,后跟  $0 \sim n$  个逗号分隔的变量指针,用以存放转换后的对象。变量指针的个数对应于格式串中以 % 打头的变量转换参数的个数。严格禁止变量指针的个数少于对应数,这时会引发灾难性后果。格式串中变量转换参数的格式如下(方括号内的域是可选的):

%[flags][width] type

其中:① [flags]② [width]③ type 的解释和内容同 E.50。

vsscanf() 与 sscanf() 相似,但是使用指针参数表而不是变量参数表。请见 va\_list()。

[例 E.56]

```
#include <stdio.h>
```

```
void main()
```

```
{ char c;
```

```
int i;
```

```
long l;
```

```
unsigned char uc;
```

```
unsigned int ui;
```

```
unsigned long ul;
```

```
float x, y;
```

```
char buf[10];
```

```
int n;
```

```
printf("Enter signed char , int, and long numbers respectively, please:");
```

```
n = sscanf("2 - 345678912", "%bd %d %d", c, i, l);
```

```
printf("There %d numbers are read. \n", n);
```

```
printf("Enter unsigned char , int, and long numbers respectively, please:");
```

```
n = sscanf("24456781234", "%bu %u %u", uc, ui, lu);
```

```
printf("There %d numbers are read. \n", n);
```

```
printf("Enter a char , and a string respectively, please:");
```

```
n = sscanf("a blacksmith", "%c %9s", c, buf);
printf("There %d arguments are read. \n", n);
```

```
printf("Enter two floating - point numbers respectively, please:");
n = sscanf("3.1415922.6", "%f %8f", x, y);
printf("There %d numbers are read. \n", n);
```

·参见

fscanf, scanf, sprintf, va \_ arg。

·返回值

提取及转换成功返回转换成功的参数个数。不成功返回 EOF。

## E.58 STRCAT

·提要

```
#include <string.h>
```

```
char *strcat(char *s1, char *s2)
```

·说明

将 s<sub>2</sub> 指向的字符串接续于 s<sub>1</sub>。返回指针 s<sub>1</sub>。

[例 E.57]

```
#include <stdio.h>
```

```
#include <string.h>
```

```
main( )
```

```
{
char buf[256], *s1, *s2;
```

```
strcpy(buf, "Start of line");
```

```
s1 = buf;
```

```
s2 = "...end of line";
```

```
strcat(s1, s2);
```

```
printf("length = %d \n", strlen(buf));
```

```
printf("string = %s \n", s1);
```

·参见

strcpy, strcmp, strncat, strlen。

·返回值

返回指针 s<sub>1</sub>。

## E.59 STRCHR

·提要

```
#include <string.h>
```

```
char * strchr(char * s, int c)
```

·说明

在 s 指向的字符串中搜索 c, 一旦发现返回其指针。否则返回 null。

[例 E. 58]

```
#include <string.h>
```

```
#include <stdio.h> /* for printf */
```

```
int main(void)
```

```
{ char *buf = "aaabbbcccd";
```

```
char *p;
```

```
int n;
```

```
    p = strchr(buf, 'b');
```

```
    if(p)
```

```
        printf("The character %c is found at the position: %d. \n", 'c', p - buf);
```

```
    else
```

```
        printf("The character wasn't found. \n");
```





```

elseif(i>0)
    printf("ABC is less than Abc\n");
else
    printf("ABC is equal toAbc\n");
}

```

·参见

strcpy, strcmp, strcat, strlen。

·返回值

$s_1$  大于、等于、小于  $s_2$  时分别返回正数、0、负数。

## E.61 STRCPY

·提要

```

#include <string.h>
int strcpy(char *s1, char *s2)

```

·说明

将  $s_2$  指向的字符串复制到  $s_1$ 。返回目标指针  $s_1$ 。

[例 E.60]

```

#include <stdio.h>
#include <string.h>

main( )
{
    char buf[256], *s1, *s2;

    strcpy(buf, "Start of line");
    s1 = buf;
    s2 = "...end of line";
    strcat(s1, s2);
    printf("length= %d\n", strlen(buf));
    printf("string= %s\n", s1);
}

```

·参见

strcpy, strcmp, strcat, strlen。

·返回值

返回目标指针  $s_1$ 。

## E.62 STRLEN

·提要

```
#include <string.h>
```

```
int strlen(char *s)
```

·说明

求  $s$  指向的字符串长度, 不包括 null 串尾符, 并返回之。

[例 E. 61]

```
#include <stdio.h>
```

```
#include <string.h>
```

```
main( )
```

```
{
```

```
char buf[256], *s1, *s2;
```

```
strcpy(buf, "Start of line");
```

```
s1 = buf;
```

```
s2 = "...end of line";
```

```
strcat(s1, s2);
```

```
printf("length = %d \n", strlen(buf));
```

```
printf("string = %s \n", s1);
```

```
}
```

·参见

无。

·返回值

返回字符串长度, 不包括 null 串尾符。

## E.63 STRNCAT

·提要

```
#include <string.h>
```

```
char *strncat(char *s1, char *s2, size_t n)
```

·说明

将  $s_2$  指向的字符串接续于  $s_1$ , 最大接续长度为  $n$ , 并加 null 符。  $s_1$  必须有足够长度。返回指针  $s_1$ 。

[例 E. 62]

```
#include <stdio.h>
```

```
#include <string.h>
```

```
main( )
```

```
{
```

```
char buf[256], *s1, *s2;
```

```
strcpy(buf, "Start of line");
```



```
s1 = buf;
s2 = "...end of line";
strncat(s1, s2, 5);
printf("length = %d \n", strlen(buf));
printf("string = \ " %s \ " \n", s1);
|
```

• 参见

strcpy, strcmp, strcat, strlen。

• 返回值

返回指针 s1。

## E.64 STRNCMP

• 提要

```
#include <string.h>
int strncmp(char *s1, char *s2, size_t n)
```

• 说明

基于 ASCII 字符集, 比较 s1 和 s2 的大小, 比较长度 n。s1 大于、等于、小于 s2 时分别返回正数、0、负数。

[例 E.63]

```
#include <string.h>
#include <stdio.h>
int main( )
{ char *buf1 = "abc", buf2 = "aBc";
  int i, n = 2;
  i = strncmp(buf1, buf2, n);
  if(! i)
    printf("Comparing the %d charaters, buf1 = buf2 \n", n);
  elseif(i > 0)
    printf("Comparing the %d charaters, buf1 > buf2 \n", n);
  else
    printf("Comparing the %d charaters, buf1 < buf2 \n", n);
  return(0);
|
```

• 参见

strcpy, strcmp, strcat, strlen。

• 返回值

s1 大于、等于、小于 s2 时分别返回正数、0、负数。

## E.65 STRNCPY

• 提要



```
#include <string.h>
```

```
int strncpy(char *s1, char *s2, size_t n)
```

·说明

将 s<sub>2</sub> 指向的字符串复制到 s<sub>1</sub>, 最长为 n 字符。s<sub>2</sub> 长于 n 则被截断并加 null 符。返回目标指针 s<sub>1</sub>。

[例 E.64]

```
#include <stdio.h>
```

```
#include <string.h>
```

```
main( )
```

```
{
```

```
char buf[256], *s1, *s2;
```

```
strncpy(buf, "Start of line", 6);
```

```
s1 = buf;
```

```
s2 = "...end of line";
```

```
strcat(s1, s2);
```

```
printf("length = %d \n", strlen(buf));
```

```
printf("string = %s \n", s1);
```

```
}
```

·参见

strcpy, strcmp, strcat, strlen。

·返回值

返回目标指针 s<sub>1</sub>。

## E.66 STRRCHR

·提要

```
#include <string.h>
```

```
char *strrchr(char *s, int c)
```

·说明

在 s 指向的字符串中, 从尾部开始搜索 c, 一旦发现返回其指针。否则返回 null。

[例 E.65]

```
#include <string.h>
```

```
#include <stdio.h> /* for printf */
```

```
int main(void)
```

```
{ char *buf = "aaabbbcccd";
```

```
char *p;
```

```
int n;
```

```
p = strrchr(buf, 'b');
```

```

if(p)
    printf("The character wasn't found. \n");
else
    printf("The character wasn't found. \n");
return(0);

```

·参见

strchr, strcmp, strncat, strlen。

·返回值

返回首先匹配处的指针。否则返回 null。

## E.67 TAN

·提要

```

#include <math.h>
double tan(double f)

```

·说明

求指定参数的正切值。用多项式展开求近似值。

[例 E.66]

```

#include <math.h>
#include <stdio.h>
#define C3.141592
main( )
{
    double i;

    for(i=0;i<180.0;i+=10)
        printf("tan( %3.0f) = %f\n", i, tan(i * C/180.0));
}

```

·参见

cos, sin, asin, acos, atan。

·返回值

$-\infty \sim +\infty$  范围的 double 数。

## E.68 TOLOWER, TOUPPER, TOASCII

·提要

```

#include <math.h>
char tolower(int c)
char toupper(int c)

```



char toascii(int c)

•说明

char tolower(int c)将参数大写字母转换为小写字母并返回。参数不是大写字母返回原参数。

char toupper(int c)将参数小写字母转换为大写字母并返回。参数不是小写字母返回原参数。

char toascii(int c)求指定参数字符的 ASCII 码, 范围为 0~127, 并返回之。

[例 E. 67]

```
#include <ctype.h>
#include <stdio.h>

void main(void)
{
    char *string = "This is a string.";
    int k, length;
    for(k = 0, k < length = strlen(string), k++)
    {
        string[k] = toupper(string[k]);
        string[k] = '\n';
        printf("Now, the string is %s\n", string);
    }

    for(k = 0, k < length, k++)
    {
        string[k] = tolower(string[k]);
        string[k] = '\n';
        printf("Now, the string is %s\n", string);
    }

    length = 0xFF41;
    k = toascii(length);
    printf("%d is converted to ascii%c", length, k);
}
```

•参见

islower, isupper, isascii。

•返回值

分别返回小写字母、大写字母、范围为 0~127 的 ASCII 码。

## E.69 VA \_ STSRT, VA \_ ARG, VA \_ END

•提要

```
#include <stdarg.h>

void va _ start(va _ list ap, last _ parmN)
type va _ arg(va _ list ap, type)
void va _ end(va _ list ap)
```

•说明

C 语言允许定义函数时在参数表中使用省略号, 它表示具有可变参数。可变参数的个数

和类型在定义时和编译时都是未知的,只有在调用时才知道具体的参数名和个数,而类型也还是不知道。文件 `stdarg.h` 提供了一个指针数组(`va_list ap`)和本节介绍的三个宏。由它们在运行时向变参数函数提供正确的参数个数和类型。`va_start(va_list ap, last_parmN)`按调用时的实参表构造变参数信息表,程序员通过 `va_arg(va_list ap, type)`的参数补充必要的类型信息。`va_end(va_list ap)`根据实参表正确地结束变参数的提取。其中:`last_parmN`是最后一个固定型参名,`type`是程序员补充的当前参数类型。

[例 E.68]

```
#include <stdarg.h>
```

```
#include <stdio.h>
```

```
int pf(int a, ... )
```

```
{
```

```
    va_list ap;
```

```
    va_start(sp, a);
```

```
    while(a--)
```

```
        puts(va_arg(ap, char *));
```

```
    va_end(ap);
```

```
}
```

```
int main( )
```

```
{
```

```
    pf(3, "line1", "line2", "line3");
```

```
}
```

•参见

无。

•返回值

`va_arg( )`返回 `type`,其他无返回。

## 附录 F XAC 使用错误信息



下面按字母顺序列出 XAC 使用错误信息。

`#define` syntax error

定义宏句法错。可能是宏名未用字母打头或形参名未用字母打头及缺少括号。

`#elif` may not follow `#else`

不能在 `#else` 块中使用 `#elif`。

`#elif` must be in a `#if`

`#elif` 应用在 `#if` 块中。

`#else` may not follow `#else`

`#else` 不能接着再用 `#else`。

`#else` must be in a `#if`

`#else` 应该用于 `#if` 块。

`#endif` must be in a `#if`

`#endif` 应该用于 `#if` 块。

`#error; *`

这是程序员故意设置的伪指令,用以测试编译时的定义错误等。

`#if ...sizeof( )` syntax error

预处理器发现 `#if` 中的 `sizeof( )` 有参数错。

`#if ...sizeof( )`: bug, unknown type code \*

预处理器求解 `sizeof( )` 的表达式时发现内部错,查类型说明符是否有错。

`#if ...sizeof( )`: illegal type combination

预处理器求解 `sizeof( )` 的表达式时发现类型说明符结合错,如 `short long int`。

`#if` bug, operand = \*

预处理器求解表达式时发现运算符错。

#if ...sizeof( ):error, no type specified

预处理器发现 #if 中的 sizeof( )有错,缺少类型。应使用简单类型或其指针。

#if ...sizeof, unkown type \*

预处理器发现 #if 中的 sizeof( )有不能识别的类型。应使用简单类型或其指针。

#if value stack overflow

预处理器为求解 #if 中的表达式时,出现为求解表达式而设立的堆栈溢出。可能带括号的子表达式过多。

#if, #ifdef, or #ifndef without an argument

预处理器伪指令 #if, #ifdef, #ifndef 都应有参数。#if 的参数应是表达式, #ifdef, #ifndef 的参数应是简单的名字。

#include syntax error

预处理器伪指令 #include 应有参数。参数是双引号或尖括号扩起的头文件名。

‘.’ expected after ‘...’

‘...’后应补‘.’。

‘case’ not in switch

‘case’情况语句不在 switch 语句中。

‘default’ not in switch

‘default’语句不在 switch 语句中。

( expected

缺少‘(’号。它应是 while, for, do, if, asm 等关键字后的第一个单词。

)expected

缺少‘)’号。

| expected

缺少‘|’号。

| expected

缺少‘|’号。

, expected



缺少‘,’号。说明表中两标识符间缺少‘,’号,或标识符前的类型名因拼错而被误解为标识符。

-s, too few values specified in \*

预处理器 -s 选项要求的值表不完整。启动编译器或 HPDXA 需要完整的值表。

-s, too many values, \* unused

预处理器 -s 选项要求值表的项目给的过多。

...illegal in non \_prototype arg list

省略号只能出现在函数原型参数表的最后一项。

: expected

此处需要冒号。

; expected

此处需要 ; 号。

= expected

此处需要 = 号。

] expected

此处需要 ] 号。

a parameter may not be a function

函数的参数表中不能有函数,可以有函数的指针。可能是丢失了 \* 号。

argument \* conflicts with prototype

所指参数与原型不符合。

argument list conflicts with prototype

参数表与原型不符合。

argumet redeclared: \*

参数表中所指参数重复说明。

argumet redeclared: \*

函数中的指定参数在多处定义。

arithmetic overflow in constant expression



常数表达式结果超范围。

array dimension on \* ignored

作为函数参数的数组名不需要带尺寸,在传送中实际上被转换成指针。

array dimension redeclared

数组的尺寸被定义过多个非 0 的值。第一次被定义为 0,可再定义一次。

array index out of bounds

数组的下标值超过定义的尺寸或为负值。

assertion

出现编译器内部错。

assertion failed: \*

出现编译器内部错。

attempt to modify const object

对象被说明为 const,就不能再赋值或修改。

auto variable \* should not qualified

自动变量不能加修饰符。

bad #if ...defined ( ) syntax

#if 表达式中的参数宏有句法错。参数宏应有单一的字母开头的名字和放在扩号内的参数。

bad -m option: \*

代码生成器对‘-m’选项有不理解的地方。

bad ‘-p’ format

给连接器的‘-p’选项格式不对。

bad -a spec: \*

给连接器的‘-a’选项规范不对。

bad -q spec: \*

编译一趟用的‘-q’选项存在类型修饰符错。

bad arg \* to tysize



这是一个不应发生的内部错。可能执行文件被破坏,应重新用磁盘安装编译器。

bad arg to e:

出现代码生成器的内部错。

bad bconfloat - \*

出现代码生成器的内部错。

bad bit expression

汇编文件位表达式错。

bad bitfield tyope

位寻址区只能使用 int 类型。

bad character in extended tekhex line \*

objtohex 的内部错。不应发生。

bad checksum secification

校验和的错误规范。

bad combination of flags

对 objtohex 的选项组合有错。

bad complex relocation

要求连接器进行复杂的定位。可能有的目标文件被破坏。

bad confloat - \*

出现代码生成器的内部错。

bad conval - \*

出现代码生成器的内部错。

bad dimensions

代码生成器发现数组长度为 0。

bad element count expr

这是中间代码错。可能文件被破坏,应重新安装编译器。

bad fixup value

汇编文件中的修正值错误。



bad gn

出现代码生成器的内部错。

bad high address in -a spec

‘-a’选项的高地址有错,使用二、八、十六进制的数应包括后缀,十进制的数不包括后缀。

bad low address in -a spec

‘-a’选项的低地址有错,包括进制的后缀不对。

bad mod ‘+’ for how = \*

内部错。

bad non\_zero node in call graph

连接器遇到调用图中底层节点调高层节点。可能发生间接递归调用,使用的编译堆栈应不支持递归调用而引起错误。

bad object code format

目标文件目标格式错。文件被截断或被破坏或不是 HI-TECH 目标文件。

bad op \* to revlog

出现代码生成器的内部错。

bad op \* to swaplog

出现代码生成器的内部错。

bad op : \*

中间代码文件出现错误,临时文件溢出(如 ramdisk)。

bad origin format in spec

-p 选项的原始格式错,进制的后缀应正确。

bad popreg \*

出现编译器的内部错。

bad pragma \*

出现代码生成器遇到不认识的 pragma 伪指令。

bad pushreg: \*

出现编译器的内部错。



bad record type \*

目标文件不是 HI-TECH 目标文件。

bad ret\_mask

出现代码生成器的内部错。

bad segment fixups

这是来自 objtohex 的模糊信息,实际中不应发生。

bad segspec \*

连接器 -g 选项不正确。应用如下格式:

-Gnxc+o 其中:n-段数,x-乘号,c-乘数,o-常数偏移量。

例:-Gnx4+16 是指定 selector 由 16 开始每次加 4,即 16,20,24...

bad size in -s option

连接器 -s 选项不正确。进制的后缀应正确。

bad size list

涉及编译器一趟的 -z 选项不正确。指定的类型大小不对。

bad storage class

存储类错误。函数内部只能用 auto 存储类,函数的参数只能用 register 存储类修饰。存储类错误将引起中间代码不正确,导致 ramdisk 溢出。

bad string \* in psect pragma

代码生成器遇到 #pragma psect 伪指令的字符串有错。应有格式:oldname=newname。

bad switch size \*

出现编译器的内部错。

bad sx

出现代码生成器的内部错。

bad u usage

出现代码生成器的内部错。

bad variable syntax

中间代码文件错,可能导致临时文件溢出(如 ramdisk)。



bad which \* after I

出现编译器的内部错。

binary digit expected

要求二进制数, 格式应为 0Bxxx, x 为 0 或 1, 如 0B0110。

bit field too large \* bits)

最大的位数应是 int 的位数。

bit number not absolute

位数应在 0~7 之间。

bit variables must be static or global

位变量只能用 static 和 global(汇编用)修饰, 在函数内也不能说明为 auto。

bug: illegal \_ macro

预处理器内部错, 不应发生。

can't be both far and near

类型不能既修饰为 near 又修饰为 far。

can't be long

只有 int 和 float 能够用 long 来修饰。

can't be register

只有函数的参数和 auto 变量能用 register 来修饰。

can't be short

只有 int 能用 short 来修饰。

can't be unsigned

没有 unsigned float。

can't call an interrupt function

中断函数只能用硬件或软件中断来调用, 但中断函数可以调用其他非中断函数。

can't create temp file \*

编译器不能创建暂时文件, 检查是否在 path 环境变量中。

超星阅读器提醒您:  
使用本复制品  
请尊重相关知识产权!

can't create xref file \*

包括交叉访问表的输出文件不能创建。

can't find space for psect \* in segment \*

命名段不能定位到指定地址,可能代码太长,由 -A 选项指定的空间不够。

can't generate code for this expression

表达式太复杂,用中间变量简化。

can't have arrays for bits

位变量不能使用指针。

can't have an array of bits or a pointer to bit

位数组和位变量指针都是非法的。

can't have array of functions

没有函数数组,但有函数指针数组。正确的函数指针数组写法为 `int (* arayname[])( )`。

can't have pointer to bit

位变量指针是非法的。

can't have unsigned and signed together

说明中不能同时有相互矛盾的 `signed` 和 `unsigned`。

can't initialize auto aggregates

函数内部不能有带初值的组合变量(如结构、数组等),但经 `static` 修饰后则可。

can't initialize arg

函数的参数不能有初值。但在调用函数时可以传初值给参数。

can't mix proto and non \_ proto args

函数的参数在说明时,要么用 ANSI 原形格式,要么用 K&R 格式,不能混用。

can't open

文件不能打开,查拼写是否有错。

can't open avmap file \*

没有产生 Avocet 格式符号文件的程序,重新安装编译器。



can't open checksum file \*

为 objtohex 指定的校验和文件不能打开, 查拼写是否有错。

can't open command file \*

指定的命令文件不能打开, 查拼写是否有错。

can't open include file \*

指定的包含文件不能打开, 查拼写是否有错。存储器不足和文件打开过多也有可能。

can't open input file \*

指定的输入文件不能打开, 查拼写是否有错。

can't open output file \*

指定的输出文件不能打开, 查拼写是否有错。

can't reopen \*

编译器不能再次打开创建的暂时文件。

can't seek in \*

连接器找不到指定的文件。

can't take address of register variable

变量修饰为 register 后, 不能再使用 & 取地址。

can't take sizeof(bit)

sizeof 不能用于位。

can't take sizeof function

sizeof 不能用于函数。

can't take this address

不能为使用 & 的表达式指定地址, 因为它不是左值。

can't use a string in an #if

预处理器不允许在 if 表达式中使用字符串。

can't get memory

连接时存储器不够用, 一般不会, 取消 TSR 再试。





cannot open include file \*

找不到包含文件, 查文件名是否正确。如系标准头文件应当放在尖括号内。

cast type must be scalar or void

抽象类型(类型强制类型)只能是标量(不是组合量)和 void 类型。

character not valid at this point in format specifier

printf( )的格式串中使用了非法字符。

char const too long

单引号内的字符不能超过一个。

close error (disk space?)

编译器关闭暂时文件时报告有错。多数是磁盘放不下该文件。如果暂时文件创建在 ramdisk 上即使是大磁盘也会发生本错误。

common symbol psect conflict; \*

共用符号段定义了不止一个。

complex relocation not supported for -r or -l options yet

连接器为复杂定位设置了 -r, -l 选项, 但是, 尚不可使用。

conflicting fnconf records

可能存在多个运行时启动程序, 检查连接器命令行的参数或 HPD 的 Object file... 菜单项是否用错。

constant conditional branch

条件语句总是走固定分支。多数是表达式括号不全或用错了括号, 使求解的值不是希望的值; 或者误写了 while(1) 或 for(;;) 形式, 造成无穷循环。

constant conditional branch; possible use of = instead of ==

条件表达式误将 = 用为 ==, 致使将常量赋给了变量造成总是走固定分支。

constant express required

要求表达式在编译时就被求解, 并得到的是常量。

constant left operand to ?

三目操作符的条件操作符? 的左操作数误为常量, 致造成总是走固定分支。



constant operand to || or &&

逻辑操作符的一个操作数是常数,查表达式是否缺少括号或用错括号。

constant relational expression

关系表达式结果总是真或假,譬如无符号数与负数相比较;变量与超范围的数相比较。

control line \* within macro expansion

宏展开中出现预处理器控制行。不应该出现。



declaration of \* hide outer declaration

当前函数定义了和其过去外层重名的对象。这是合法的。但是,如想用此名访问外层变量时则发生意外。

declarator too complex

说明过于复杂,超越了编译器的能力范围,应设法简化。编译器尚且认为复杂,对维护的人员就更不用说了。

default case redefined

switch 语句只允许有一个 default 标号,现在多于一个。

degenerate signed comparision

符号数和最小的负数相比较,致使总是真或伪。如:

char c;

if(c >= -128)

if 的条件表达式总是真。

degenerate unsigned comparision

无符号数与 0 相比较有与上条相似的退化现象。

dimension required

只有最高维数的大小可以不做指定,其余维数的大小都要指定。

divide by zero in #if, zero result assumed

if 的条件表达式中有 0 除,其结果被定为 0。

division by zero

常量表达式中有 0 除。

double float argument required

printf()的格式串的%f 参数要求浮点数,查有无缺少或多余的参数。

duplicate -m flag

-m 标志重复。连接器只能有一个-m 标志,除非其中有一个未指定映像文件。

doublicate case label \*

switch 语句中,有多个标号对应于一个 switch 值。

duplicate label \*

函数中标号重名。注意,函数中的标号、作用域是整个函数而不是标号所在的块。

duplicate qualifier

修饰符被多次说明。有两处可以为类型指定修饰符:直接定义和使用 typedef。把多余的修饰符去掉。

duplicate qualifier key \*

通过-q 又指定了一次同一个修饰符。

duplicate qualifier name \*

通过-q 又指定了一次同一个修饰符。

end of file within macro argument from line \*

宏参数缺少结束符,可能是缺闭圆括号。行号指的是参数的开始。

end of string in format specifier

printf()的格式串使用不当。

entry point multiply defined

送给连接器的目标文件中有不止一个的入口点。

enum tag or } expected

关键字 enum 之后应后跟标识符作为 enum 的标桩,或是后跟'}'。

eof in #asm

在 #asm 块中出现 eof,可能遗忘了 #endasm 或 #endasm 拼错。

eof in comment

注释中遇到 eof,查注释有否结束符。



eof on string file

编译一趟到存放常量字串的文件中读字串时,遇到 eof。多数是磁盘容量不足,查磁盘自由空间或 ramdisk 的大小。

error in format string

printf( ) 的格式串的使用不当。如果运行,会产生意想不到的结果。

evaluation period has expired

编译器软件的评估期已过,请与 HI-TECK 联系。

expand - bad how

出现代码生成器的内部错。

expand - bad which

出现代码生成器的内部错。

expected ' - ' in -a spec

在 -a 选项的低高地址之间应有 ' - ' 号,如:

- AROM = 1000h - 1fffh

expoent expected

浮点常数的指数 e(E) 部分至少应有一位数字。

expression generates no code

表达式未产生代码。查是否缺少圆括号等。

expression stack overflow at op \*

# if 表达式求解时要使用 128 字节的堆栈,表达式过繁堆栈会溢出。

expression syntax

表达式格式不对。

expression too complex

表达式过繁编译器内部堆栈会溢出。应分解表达式。

external declaration inside function

函数内部存在带 external 的说明。这是合法的,但并非好事。它限制说明对象的作用域局限于函数体。这意味着编译器在同一文件的以后再遇见同名对象的引用或定义性说明时就不再理会它的存在,因而导致程序行为离奇和连接时报签字错误。并且,对以前同名对象的说明实



行隐蔽,暗中搅乱编译器对于类型的检查。作为通用的规则,将变量和函数说明为 `extern` 时一定要放在任何函数的外面。

field width not valid at this point

`printf()` 格式串中关于类型的宽度有错。

filename work buffer overflow

创建文件名时,要使用内部长度为 4 096 字节的缓冲器对包含文件进行扫描。现在,此缓冲器溢出。一般这是不应发生的。

fixup overflow referencing \*

要求连接器给某项分配的地址不合适,发生超范围引用,如字节范围的对象分配的起始地址大于 255。

fixup overflow in expression \*

要求连接器分配给某段的地址超出能用的范围,如给字节范围的对象分配了大于 255 的起始地址。此类错误多发生在复杂表达式中。

float param coerced to double

对 K&R 风格的参数说明,根据缺省规定将浮点参数一律转换为 `double`。如果采用这种风格的说明,应该坚持这种风格,不应该和 ANSI C 风格的函数混用。

formal parameter expected after #

在字符转换符 '#' 的后面,应该使用参数宏的形式参数。

function \* appears in multiple call graphs: rooted at \*

函数在调用图中见到既被 `main` 又被中断函数调用。解决办法:如果编译器支持本函数可重入,则将函数修饰为 `reentrant` 函数;或重新改写成无局部变量和参数的函数;或写成两个不同的函数,分开调用。

function \* is never called

函数从未被调用。如应被调用,则查错;如不应被调用,则删之,以省空间。

function body expected

当函数的参数具有 K&R 风格时,要求后跟函数体。

function declared implicit int

编译器遇到对未曾定义函数的调用,将自动隐式地生成 `int` 类型无参数的函数说明(K&R 格式)。假如后来遇到了此函数的定义,但与隐式生成的函数不符,则会给出编译错。为保证不

出此类问题,一定要保证定义先于调用,或使用 ANCI C 风格的说明,并在调用之前给出原型说明,且在说明的最前面加上 `extern` 或 `static`。

function does not take arguments

调用没有参数的函数加上了参数。

function is always "extern", can't be "static"

函数已经说明为 `extern`(包括隐式说明)不能再加 `static`。后加的 `static` 无效。问题发生在超前引用的,可把引用说明中的“`extern`”改为“`static`”;或者直接把带 `static` 的定义性说明放到前面。

function or function pointer required

函数调用可以使用函数名或函数指针。当变量或表达式句法有错而出现后跟开圆括号时,会被误解为企图调函数。这时会报本错误。

function can't return arrays

函数只能返回标量类型或结构,数组不行。

function can't return functions

函数不能返回函数但能返回函数指针,这时有类似下面的形式:

```
int( *(函数名( ))( )
```

function nested too deep

本错误不发生在 C 语言中,因 C 的函数不允许嵌套函数。

hex digits expected

0x 后面至少应有一位十六进制数字。

ident records do not match

传给连接器的目标文件中存在别的处理器的目标文件。

identifier expected

‘enum’说明的大括号内必须有被逗号分隔的标识符。

identifier redefined: \*

标识符已经定义。

identifier redefined: \* (from line \*)

标识符已经定义。from line \* 是上次的定义处。



illegal # command \*

非法预处理器控制行,可能预处理器关键字有错。

illegal # if line

# if 表达式有错。

illegal # undef argument

# undef 的参数应是有效的名字。

illegal ' #' directive

非法带 # 的伪指令。

illegal - o flag

非法 - o 编译选项,其后应紧跟文件名,如: - o file.obj。

illegal - p flag

非法 - p 编译选项。

illegal character \*

非法字符。

illegal character \* (decimal) in # if

# if 表达式有非法十进制数字字符。

illegal character \* in # if

# if 表达式有非法字符。

illegal conversion

表达式要求进行不兼容的类型转换,如结构转 int。

illegal conversion between pointer tyoes

非法指针类型转换,可能是用错了变量名。如果是有意转换指针类型,应加指针强制。

illegal conversion of integer to pointer

非法地将 int 指派给指针或转换为指针,可能用错变量名。如果是有意进行的应加类型强制。

illegal conversion of pointer to integer

非法地将指针指派给 int 或转换为 int,可能用错变量名。如果是有意进行的应加类型强制。

illegal flag \*



非法标志。

illegal function qulifier(s)

给函数指定了非法的修饰符,如常 const、volatile 等。这些修饰符是用于左值以规定其存储空间。这样的修饰符可能是丢失了 \* 号的函数返回指针所需要的。

illegal initialization

不能对 'typedef' 进行初始化,因为它并未分配存储器。

illegal operator in #if

#if 表达式存在非法运算符。

illegal opreration on a bit variable

不是所有的操作符都支持位变量。

illegal or too many -p flag

连接器选项 -p 使用过多,将它们加以合并。

illegal record type

目标文件有错。可能目标文件给的不对或发生连接器内部错。重新建立目标文件。

illegal relocation size: \*

连接器读到的目标文件有错。可能是连接器过时或连接器、汇编器有内部错。

illegal relocation type: \*

目标文件中存在含有错误的可再定位的记录。可能是文件被破坏或不是目标文件。

illegal type for array dimmension

数组的大小必须是 int 或枚举量。

illegal type for index expression

下标表达式必须是 int 或枚举量。

illegal type for switch expression

'switch' 表达式必须是 int 或枚举量。

illegal use of void expression

void 表达式是没有值的,所以不能用做操作数。





image too big

objtohex 产生的映像文件过长, 虚拟内存不足。

implicit conversion of float to integer

不能隐式地要求浮点数转换为 int 类型, 如打算截短浮点数, 应改为显式强制。

implicit return of non\_void function

说明为有返回的函数, 在代码中存在着隐式的无返回的路径。

implicit signed to unsigned conversion

符号数向高级无符号数的隐式转换是按照保值的原则进行的, 先保值扩大再转为无符号数。这时, 出现不希望的符号扩展。为避免这种符号扩展, 应先显式地强制转换为无符号数, 然后再扩大。

inappropriate 'else'

else 没有与之配套的 if。

inappropriate break/continue

break, continue 语句与所在的控制结构不配套。continue 只能用于 while, for, do while 循环结构中, 而 break 还可以用于 switch 语句中。

inappropriate intermediate code version ; should be \*

编译一趟生成的中间文件与代码生成器的版本号不兼容。可能是有不相兼容的多个版本编译器放在同一目录中, 或暂时文件被破坏所引起的。检查 TEMP 环境变量, 如果路径名较长, 改短再试。

incomplete \* record body: length = \*

目标文件中包含有长度不对的记录, 可能文件被截短或不是目标文件。

incomplete record

传给 objtohex 的目标文件被破坏。

incomplete record : \*

目标代码文件不完整, 可能已被破坏或目标文件不对。重新编译源文件, 观察是否出现磁盘空间不足。

incomplete record : typoe = \* length = \*

目标代码文件不是 HI\_TECK 目标文件或被因 randisk 空间不足而被截短。

inconsistent storage class



说明的存储类自相矛盾。

inconsistent type

一个说明存在了两个类型。

initialization syntax

体中句法有错, 检查括号和逗号的用法。

initializer in 'extern' declaration

使用 extern 的说明不能有初值。

integer constant expected

结构成员名后面跟有冒号的是位域。位域要求用整型数指明该域的位数。

integer expression required

在枚举说明中可给成员赋值, 但表达式的结果应是整型常量。

integer argument required

格式串要求整型参数, 检查参数个数和次序。

integer type required

操作符只接受整型数。

invalid disable: \*

预处理器内部错, 不应发生。

invalid format specifier and type modifier

printf 的格式串不对。

label identifier expected

goto 之后必须有标号。

invalid qualifier combination on \*

修饰符不能组合使用。

label not followed by :

标号缺少冒号。

library is badly ordered

库的次序安排得不好,虽然能够连接,但影响速度。

line does not have a new line on the end

文件的末尾缺少换行符。有些编辑器生成这种文件,但它使包含文件发生问题。ANSI C 要求所有源文件中使用的都是整行。

local psect ' \* ' conflicts with global psect of same name

局部 psect 和全局 psect 发生同名的矛盾。

logical type required

if, while 语句的表达式和 boolean 运算符(如!, && 等)都要求整型的标量类型。

long argument required

格式串要求长整型参数,检查参数个数和次序。

macro was 't defined

-u 指定的预处理器宏没有定义,所以 undef 不能进行。

macro work area overflow

宏展开超过了内部开辟的用于宏展开的表的长度 8 192 字节(8K)。

member \* redefined

本结构或联合中已经用过了这个成员名。

member can't functions

结构或联合的成员不能用函数,但可用函数指针:如 'int( \* anme)( );'。

metregister \* can't be used directly

出现编译器内部错。

mismatched comparision

变量或表达式与常量进行比较,但是所给常量超范围,致使比较结果永远是真或是假,如 unsigned char 与 300 相比就是这样。

misplced '?' or ':', previous oprator is \*

在 #if 表达式中遇到 ':' 操作符,但是找不到与之配套的操作符 '?',检查括号等。

misplaced constant in #if

#if 表达式中常量位置不对,可能缺少某个操作符。

missing ')'

表达式缺少')'。

missing '=' in class spec

段的类别选项部分缺少=号,如 - Ctext = ROM。

missing ']'

表达式缺少']'。

missing arg to -a

选项 -a 要求修饰符作为参数。

missing arg to -u

选项 -u 要求参数,如 -U \_symble。

missing arg to -w

选项 -w 要求数字作为参数。

missing argument to 'psect'

#pragma psect 要求 oldname = newname 作为参数,其中:oldname 是编译器认识的以存在的 psect 名,newname 是欲用的新名,如:

#pragma psectbss = battery

missing basic type: int assumed

说明未包括基本类型,已经假定为 int。虽不是非法,但是不好。

missing key in avmap file

要求产生 Avocet 符号文件,做不起来。重新安装编译器。

missing memory key in avmap file

要求产生 Avocet 符号文件,做不起来,重新安装编译器。

missing name after pragma 'print \_check'

#pragma print \_check 要求函数名作为参数,以便对它进行格式串的检查。

missing number after \* in -p option

-p 选项的 \* 号后面缺少数字。

missing number after % in -p option



超星数字图书馆  
使用本复制品  
请尊重相关知识产权!

-p 选项的 % 号后面缺少数字。

missing number after pragma 'pack'

#pragma pack 要求数字作为参数, 以确定结构内部成员的对齐规则, 如 #pragma pack(1) 为字符界对齐。应注意有的微处理器(如 68000, 8096)对于字强制向偶地址对齐, 如果说明为字符界对齐, 运行将不正常。

mode by zero in #if, zero result assumed

取模操作若被 0 除, 则将模设定为 0。

moudule has code belowe file base of \*

-C 选项已经指定了二进制输出文件的映像地址, 但是本模块却在此地址之后有代码。这说明本模块应该在此二进制输出文件的前面。检查汇编文件有否丢失 psect 伪指令。

multi-byte constant \* isn't portable

多字节常量是不能被移动的, 但实际上后趟编译须丢弃它们。错误来自预处理器。

multiple free: \*

出现编译器的内部错。

nested #asm directive

#asm 伪指令不允许嵌套。检查 #endasm 有否丢失或拼错。

newted comments

出现嵌套的注释。发生于结束符丢失或写错。

no #asm before #endasm

#endasm 前面没有 #asm。

no end record

目标文件没有结束记录, 可能被破坏或不是目标文件。

no end record found

目标文件没有结束记录, 可能被破坏或不是目标文件。

no identifier in declaration

说明中缺少标识符。编译器被缺少括号所困惑时, 也会发出这个错误。

no memory for string buffer

编译一趟对字符串排序和合并时, 不能为长串安排存储器。要减少模块的字符串和其长度。



no psect specified for functionvariable/argument allocation

缺少应分配给函数变量的 psect。这可能是缺少了运行时启动模块的缘故。查连接命令行的参数或 HPD 的 Object file...

no start record: entry point default to zero

传给连接器的目标文件中,找不到哪个文件有起始记录。虽然按照缺省已给程序指定起始地址为 0,但这是危险的。建议用启动模块的 END 伪指令指定起始地址。

no. of arguments redeclared

函数的参数数量与说明不符。

nodecount = \*

出现代码生成器的内部错。

non-prototyped function declaration: \*

函数使用的是老式的参数说明(K&R 风格),最好使用函数原型说明。如函数没有参数应写成:intfunc(void)。

non-acalar types can't econverted

结构、联合、数组不能转换为其他类型。但它们的指针可以转换为其他类型。本错可能是缺少 '&' 所致。

non-void founction returns no value

定义的是有返回值的函数,但是却没有带返回值的 return 语句。

not a member of the struct/union \*

标识符不是结构/联合中的成员。

not a variable identifier: \*

不是变量的标识符,可能是标号等对象。

not an argument: \*

老式的参数说明(K&R 风格),参数的类型说明不在括号内。

nxtuse( ): unknown op \*

内部错。

object file is not absolute

目标模块的目标代码版本高于联结器可操作目标代码的最高版本。使用正确的联结器。



only functions may be qualified interrupt

修饰符 interrupt 只能用于函数。

only functions may be void

变量不能使用 void。函数可以使用 void。

only lvalues may be assigned to or modified

只有左值(可寻址变量或表达式)才能被赋值或被修改。被类型强制过的左值不再是左值。用某一变量的地址存放其他类型的值是可以的,但要先经指针转换类型再用指针间接访问,这时所得的结果仍是左值,如:\*(int\*)&x=2;但:(int)x=2;非法。

only modifier l valid with this format

本格式的修饰符只有 l(long)是合法的。

only modifier h and l valid with this format

对于 printf()的格式串使用修饰符 h 和 l 是合法的。

only register storage class allowed

对于函数的参数只有 register 是可用的存储类。

operands of \* not same type

二目操作符的操作数是不同的指针类型。应加类型强制才可使用。

operator \* in incorrect context

#if 表达式中的操作符用得不对,如两个双目操作符之间没有操作数。

out of memory for strings

优化程序受限于分配的缓冲器容量,因放不下字符串而不能进行下去。

out of far memory

编译器远存储容量不足。取消 TSR,或用 EMS。

out of far memory(wanted \* bytes)

代码生成器不能分配更多的远存储器。取消 TSR 等。

out of near memory

近存储器不够编译器使用。可能符号太多,去掉头文件中未用的符号,或将文件分小。

out-of-range case label \*



switch 语句的控制表达式不会产生与此情况标号相配套的值。

out of space in macro \* arg expansion

参数宏超长,使内部缓冲器(4 096 字节)不足。

output file cannot be also an input file

要求输出文件向输入文件之一写入,这是不允许的,因为连接器对输入输出文件须同时进行读写。

pointer required

此处要求指针。

pointer to \* argument required

格式串要求指针参数。查格式串的参数数目和顺序。

pointer to non - static object returned

函数返回的指针指向非静态变量。这是不允许的,因为自动变量在函数返回后已不存在。

portion of expression had no effect

部分表达式对求值不起作用。

possible pointer truncation

对缺省指针或 near 指针又修饰为 far;或对缺省指针又修饰为 near,根据所用的存储模式,会发现有的指针可能被截短,从而丢失信息。

preprocessor assertion failure

预处理器伪指令 # assert 的参数求解后为 0。这是程序员的错误。

probable missing '}' in previous block

编译器发现好象是有函数丢了 '}'。

psect ' \* ' re - orged

本 psect 被定位了多次,查连接器选项。

psect \* cannot be in classes \*

psect 只能归属于一个类别。可能汇编的 class = 选项与连接器的选项 - C 相矛盾。

psect \* cannot be in classes \* and \*

psect 只能归属于一个类别。

psect \* in more than one group

psect 只能归属于一个 group。

psect \* not loaded on \* boundary

给有再定位边界要求的 psect 指定了不当的起始地址,如给要求定位边界为 4K 字节的 psect,用选项 -p 指定了 100H 的起始地址。

psect \* not reloaded on \* boundary

给有再定位边界要求的 psect 指定了不当的起始地址,如给要求定位边界为 4K 字节的 psect,用选项 -p 指定了 100H 的起始地址。

psect \* not specified in -p option

在连接器 -p 选项中未指定本 psect 的起始地址,虽然已连接在程序的最后,但是可能不是原意。

psect \* not specified in -p option(first appears in \*)

在连接器 -p 选项中未指定本 psect 的起始地址,在编译器 -A 选项中也无本 psect 的地址,虽然已连接在缺省位置,但是可能不是原意。

psect \* re-orged

重复指定了 psect 的起始地址。

psect \* selectorvalue redefined

重复指定 psect 的 selector 值。

psect \* type redefined \*

psect 在不同的模块中定义了不同的类别。可能是企图将不相兼容的模块进行连接。

psect exceeds max size: \*

psect 超长。

psect is absolut: \*

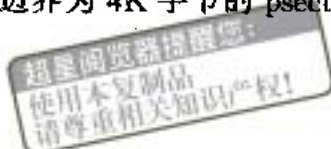
psect 是绝对段,应在 -p 选项中指定地址。

psect origin multiply defined: \*

psect 的起始地址被指定多次。

psect property redefined

psect 的属性指定多次而又不同。



psect selector redefined

psect 的 selector 指定多次而又不同。

qualifier redefined

psect 的修饰符指定多次而又不同。

read error on \*

连接器读本文件时出错。

record too long

目标文件是非 HI-TECH 有效目标文件。

record too long: \*

目标文件具有非法长度的记录,可能文件被破坏或不是目标文件。

recursive function calls

函数自身或之间发生递归调用。其中有的函数存在分配于编译堆栈内的局部变量。改用可重入函数或改写程序避免递归调用。

recursive macro defination of \*

宏展开时发生自我递归展开。

redefining macro \*

宏被重新定义为另外的内容,如果真想这样做,要先用 #undef 解除原来的宏定义。

redundent & applied to array

对数组名使用地址操作符 &,这是多余的,已被删除。

regused - bad arg to g

内部错。

relocation offset \* out of range \*

目标文件包含偏移量进入前段记录的可再定位段,原因可能是目标文件被破坏。

seek error: \*

连接器要写时找不到输出文件。

segment \* overlaps segment \*

命名段覆盖了代码段或数据段。查 -p 选项分配的地址。



signatures do not match

指定的函数在不同的模块中有不同的签字。很可能是在一个模块中错给了两个原型说明。查这两个模块各自所见的原形是什么样的,并使它们一致起来。

signed bitfield not supported

仅支持无符号位域。如果给出 int 类型的位域,编译器将其转换为无符号数。

simple integer expression required

操作符@后要求放简单的 int 表达式作为相关连的绝对地址。



simple type requires for \*

本操作符要求简单类型的操作数。

sizeof external array \* is zero

sizeof()对外部数组的求值为 0。说明外部数组时并不要求显式地给出其尺寸。

sizeof yields 0

代码生成器曾对某对象求值发现为 0。一般说来,指向数组的指针用途不大。如果需要使用指向不知尺寸的数组的指针,那就定义一个指向单一元素的指针,然后用增量或下标变量指向各元素。

storage class illegal

不能为结构/联合的成员指定存储类。它们的存储类取决于结构/联合。

storage class required

变量或函数重复说明了不同的存储类。它们可能来源于两个矛盾的说明或有一个是隐含产生的。

strange character \* after ##

单词接续符##接续的不是字母或数字。

strange character \* after # \*

#后是不希望的字符。

string expected

asm 的操作数应该是括在括号内的字符串。

struct/union member expected

结构/联合的成员名必须跟在'.'或'-'之后。

struct/union member redefined: \*

结构/联合重复定义。

struct/union member required

‘.’之前要求结构/联合名。

struct/union tag or ‘|’ expected

关键字 struct/union 之后应该是表明 struct/union 的标识符或‘|’。



symbol \* can't be global

目标文件有错, 局部符号被说明为全局, 不是无效文件就是连接器内部错。重新生成目标文件。

symbol \* has erroneous psect

目标文件有错, 符号出现在无效 psect 内。不是无效文件就是连接器内部错。重新生成目标文件。

symbol \* not defined in #undef

作为 #undef 参数的符号以前没有定义, 仅给出警告。最好使用:

#ifdef...

#undef...

#endif

syntax error in -a spec

-a 格式错, 应如: -AROM = 1000h - 1fffh

syntax error in checksum list

连接器读校验和表时有错。校验和表是按照选项由标准输入读进的, 重读。

text do not start at 0

代码未从 0 开始。

text offset too low

这个错误不大会发生。

text record has bad length: \*

目标文件有错。不是无效文件就是连接器内部错。重新生成目标文件。

text record has length too small: \*

目标文件不是 HI-TECH 有效的目标文件。

this function too large - try reducing level of optimization

使用 -Og(全局优化)选项时,遇到过长的函数。去掉全局优化选项重新编译或将函数改小。

this is a struct

在关键字 struct/union/enum 之后的标识符已成为标桩,此标桩只能和关键字 struct/union/enum 连用。

this is a union

在关键字 struct/union/enum 之后的标识符已成为相应的标桩,此标桩只能和关键字 struct/union/enum 连用。

this is an enum

在关键字 struct/union/enum 之后的标识符已成为相应的标桩,此标桩只能和关键字 struct/union/enum 连用。

too few arguents

所给函数的参数过少。

too few arguents for format string

所给格式串的参数过少,运行中会转换出一些垃圾并显示出来。

too many ( \* ) enumeration constants

所给枚举类型的常量过多。极限为 512。

too many ( \* ) structure members

所给 struct/union 成员过多。极限为 512。

too many arguments

所给函数的参数过多。

too many arguments for format string

所给格式串的参数过多,虽无害但不对。

too many arguments for macro

所给宏的参数过多。按照 ANSI C 最多为 31 个。

too many arguments in macro expansion

所给宏的参数过多。按照 ANSI C 最多为 31 个。



too many comment lines \_ discarding

编译器生成的汇编码中嵌入了远比源语句的注释行多得多的注释。为使优化时不会感到内存不足而将它们删掉。

too many file arguments. Usage:cpp[input[output]]

C 预处理器执行时最多只能给两个文件。

too many include directories

为预处理器寻找包含文件最多可以指定 7 个目录。

too many initializers

所给初值过多。

too many nested # \* statments

# if, # ifdef 等块最多嵌套 32 层。

too many operands

所给指令的操作数过多。

too many psect class specifications

所给 psect 的类别过多(-C)。

too many psect pragmas

所给 # pragma psect 伪指令过多。

too many psects

psect 段用得过多。

too many qualifiers

所给修饰符过多。

too many relocation items

objtohex 已将表用尽。程序过于复杂。

too many segment fixups

目标文件要交给 objtohex 照应的段太多。

too many segments

目标文件要交给 objtohex 照应的段太多。



too much indirection

指针嵌套最多 16 层。

too much pushback

为预处理器内部错。

type conflict

操作符的操作数的类型不相容。

type modifier already specified

类型修饰符已经指定。

type modifiers not valid with this format

本格式无需类型修饰符。

type redeclared

函数或变量重复定义。可能两个说明(其中有一个可能是隐含说明)不兼容。

type specifier reqd. for proto arg

原型参数要求类型说明符, 光有标识符不行。

unbalanced paren's, op is \*

# if 表达式的括号不配对。

undefined enum tag: \*

未定义枚举的标桩。

undefined identifier: \*

符号在程序中使用, 但未定义或说明。

undefined shift \* bits

某数欲移位的位数等于或大于该数类型的位数。这时, 对于许多微处理器将为不定。

undefined struct/union

未定义 struct/union 的标桩。检查有无拼错。

undefined struct/union: \*

未定义 struct/union 的标桩。检查有无拼错。





undefined symbol \* in #if, 0 used

#if 表达式是未定义的宏,从表达式的角度出发,已经将它置为 0。

undefined symbol:

连接时,指定的符号未被定义,可能拼错或是有未被连入的模块。

undefined symbols:

连接时,指出的符号表未被定义。

undefined variable: \*

变量在用,但截止当前未被定义。

unexpected in #if

#if 语句存在不应有的反斜杠。

unexpected end of file

可能是文件因磁盘(或 ramdisk)容量不足而被截断。

unexpected eof

遇到不应有的 eof,查句法。

unexpected text in #control line ignored

#控制行的后面有多余的字符,认为是缺少注释符的注释,已将其丢弃并给出报警,如: #endif something 则将 something 丢弃。

unknown complex operator \*

目标文件出现错误。不是无效目标文件就是连接器内部错,重新创建目标文件。

unknown fnrec type \*

不是有效的 HI-TECH 目标文件。

unknown option

不认识的预处理器选项。

unknown pragma

不认识的预处理器的 pragma 控制。

unknown psect: \*

本 psect 在 -p 选项中被列出,但是程序的任何模块中未定义。

unknown qualifier \* given to -a

编译一趟的 -a 选项, 要求已定义过的修饰符作为它的参数。

unknown record type: \*

连接器读到是无效的目标文件, 不是文件被破坏就是非目标文件。

unknown symbol type \*

连接器不认识符号的类型, 查所用连接器是否正确。



unreachable code

此代码段从未被执行, 因为没有能够接近它的路径。查控制结构(如 while, for...)内部是否有丢失的 break 语句。

unreachable matching dash \_\_\_\_\_

此结构成员从未用过,可能是多余的。

unused structure: \*

此结构标桩从未用过,可能是多余的。

unused typedef: \*

此 typedef 从未用过,可能是多余的。

unused union: \*

此 union 类型从未用过,可能是多余的。

unused variable declaration: \*

此变量从未用过,可能是多余的。

unused variable definition: \*

此变量从未用过,可能是多余的。

variable may be used before set: \*

变量可能在使用前尚未被赋值。因是自动变量定义时其值是随机数。

void function cannot return value

void 函数不能有返回值。所有 return 语句都不能有后跟表达式。

while expected

在 do 语句的最后应有 while 关键字。

work buffer overflow \*

预处理器的内部缓冲器(4 096 字节)已满。

write error (out of disk space?)

可能硬盘或 ramdisk 已满。

write error on \*

命名文件发生写错误。可能硬盘或 ramdisk 已满。

wrong number of macro arguments for \* - instead of \*

使用宏时参数数目给错。





Powered by xiaoguo's publishing studio  
QQ:8204136